ТОМАС КОРМЕН
ЧАРЛЬЗ ЛЕЙЗЕРСОН
РОНАЛЬД РИВЕСТ
КЛИФФОРД ШТАЙН

АЛГОРИТМЫ

ПОСТРОЕНИЕ И АНАЛИЗ

ТРЕТЬЕ ИЗДАНИЕ





Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

INTRODUCTION TO
ALGORITHMS

THIRD EDITION

The MIT Press
Cambridge, Massachusetts London, England

Томас Кормен
Чарльз Лейзерсон
Рональд Ривест
Клиффорд Штайн

АЛГОРИТМЫ

ПОСТРОЕНИЕ И АНАЛИЗ

ТРЕТЬЕ ИЗДАНИЕ



Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26–018.2.75
А45
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С. Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И. В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Кормен, Томас Х. и др.

А45 Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. — М. : ООО "И. Д. Вильямс", 2013. — 1328 с. : ил. — Парал. тит. англ.

ISBN 978–5–8459–1794–2 (рус.)

ББК 32.973.26–018.2.75

Научно-популярное издание

Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн

Алгоритмы: построение и анализ

3-е издание

Литературный редактор *Л. Н. Красножон*
Верстка *А. Н. Полинчик*
Художественный редактор *Е. П. Дынник*
Корректор *И. В. Красиков*

Подписано в печать 31.06.2013. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 125,1. Уч.-изд. л. 96,2.
Тираж 1500 экз. Заказ № 3726.

Первая Академическая типография "Наука", 199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978–5–8459–1794–2 (рус.)
ISBN 978–0–2620–3384–8 (англ.)

Оглавление

Содержание

Предисловие

Вначале были компьютеры, но перед компьютерами были алгоритмы. Теперь же, когда есть множество компьютеров, есть еще больше алгоритмов, и алгоритмы лежат в основе вычислений.

Эта книга служит исчерпывающим вводным курсом по современным компьютерным алгоритмам. В ней представлено большое количество конкретных алгоритмов, которые описываются достаточно глубоко, однако таким образом, чтобы разработка и анализ были доступны читателям всех уровней подготовки. Мы старались обойтись элементарными пояснениями, но при этом не нанести ущерба ни глубине изложения, ни математической строгости.

В каждой главе представлен определенный алгоритм и описаны метод его разработки, область применения и другие связанные с ним вопросы. Алгоритмы описываются и простым человеческим языком, и с помощью псевдокода, разработанного таким образом, чтобы быть понятным любому, у кого есть хотя бы минимальный опыт программирования. В книге представлены 244 рисунка, иллюстрирующих работу алгоритмов, и многие из них состоят из нескольких частей. Поскольку один из важнейших критериев разработки алгоритмов — их *эффективность*, каждое описание алгоритма включает в себя тщательный анализ времени его работы.

Данный учебник предназначен, в первую очередь, для студентов и аспирантов, изучающих тот или иной курс по алгоритмам и структурам данных. Он также будет полезен для технических специалистов, желающих повысить свой уровень в этой области, поскольку описание процесса разработки алгоритмов сопровождается изложением технических и математических вопросов.

В этом, третьем, издании книга вновь существенно изменена. В ней появились новые главы, пересмотренный псевдокод и более активный стиль изложения.

Преподавателю

Мы составляли эту книгу так, чтобы разнообразие рассматриваемых в ней тем сочеталось с полнотой изложения. Она будет полезной при чтении разнообразных курсов — от курса по структурам данных для студентов до курса по алгоритмам для аспирантов. Поскольку в книге намного больше материала, чем необходимо для обычного курса, рассчитанного на один семестр, можно выбрать только тот материал, который точнее всего соответствует курсу, который вы собираетесь преподавать.

Курсы удобно разрабатывать на основе отдельных глав. Книга написана так, что ее главы сравнительно независимы одна от другой. В каждой главе материал изложен по мере его усложнения и разбит на разделы. В студенческом курсе можно использовать только сравнительно легкие разделы, а в аспирантском — всю главу в полном объеме.

В книгу вошли 957 упражнений и 158 задач. Упражнения даются в конце каждого раздела, а задачи — в конце каждой главы. Упражнения представлены в виде кратких вопросов для проверки степени освоения материала. Одни из них просты и предназначены для самоконтроля, в то время как другие — посложнее и могут быть рекомендованы в качестве домашних заданий. Решение задач требует больших усилий, и с их помощью часто вводится новый материал. Обычно задачи сформулированы так, что в них содержатся наводящие вопросы, помогающие найти верное решение.

Отступив от принятой в предыдущих изданиях практики, мы сделали общедоступными решения ко многим, но не ко всем, задачам и упражнениям. На веб-сайте `http://mitpress.mit.edu/algorithms/` имеются ссылки на эти решения. Вы можете зайти на сайт, чтобы узнать, есть ли решение тех задач, которые вы планируете задать студентам. Ожидается, что со временем набор представленных решений будет понемногу расти, так что мы рекомендуем посещать сайт при каждом очередном чтении вашего курса.

Разделы и упражнения, которые больше подходят для аспирантов, чем для студентов, обозначены звездочкой ($\star$). Они не обязательно сложнее тех, возле которых звездочка отсутствует; просто для их понимания может понадобиться владение более сложным математическим аппаратом. Для того чтобы справиться с упражнениями со звездочкой, может понадобиться более основательная подготовка или неординарная сообразительность.

Студенту

Надеемся, что этот учебник станет хорошим введением в теорию алгоритмов. Авторы попытались изложить каждый алгоритм в доступной и увлекательной форме. Чтобы облегчить освоение незнакомых или сложных алгоритмов, каждый из них описывается поэтапно. В книге также подробно объясняются математические вопросы, необходимые для понимания проводимого анализа алгоритмов. Для тех читателей, которые уже в некоторой мере знакомы с какой-то темой, материал глав организован таким образом, чтобы эти читатели могли опустить вводные разделы и перейти непосредственно к более сложному материалу.

Книга получилась довольно большой, поэтому не исключено, что в курсе лекций будет представлена лишь часть изложенного в ней материала. Однако авторы попытались сделать ее такой, чтобы она стала полезной как сейчас в качестве учебника, способствующего усвоению курса лекций, так и позже, в профессиональной деятельности, в качестве настольного справочного пособия для математиков и инженеров.

Ниже перечислены необходимые предпосылки, позволяющие освоить материал этой книги.

- Читатель должен обладать некоторым опытом в программировании. В частности, он должен иметь представление о рекурсивных процедурах и простых структурах данных, таких как массивы и связанные списки.
- Читатель должен обладать определенными математическими навыками, в особенности навыками доказательства теорем методом математической индукции. Для понимания некоторых вопросов, изложенных в этой книге, потребуется умение выполнять некоторые простые математические преобразования. Помимо этого, в частях I и VIII рассказывается обо всех используемых математических методах.

Мы получили множество просьб предоставить решения к задачам и упражнениям из книги. На нашем веб-сайте `http://mitpress.mit.edu/algorithms/` имеются ссылки на решения некоторых задач и упражнений, так что вы запросто можете сравнивать собственные решения с нашими. Однако мы просим вас *не* присылать нам свои решения.

Профессионалу

Широкий круг вопросов, которые излагаются в этой книге, позволяет говорить о том, что она станет прекрасным учебником по теории алгоритмов. Поскольку каждая глава является относительно самостоятельной, читатель сможет сосредоточить внимание на вопросах, интересующих его больше других.

Основная часть обсуждаемых здесь алгоритмов обладает большой практической ценностью. Поэтому не обойдены вниманием особенности реализации алгоритмов и другие инженерные вопросы. Часто предлагаются реальные альтернативы алгоритмам, представляющим преимущественно теоретический интерес.

Если вам понадобится реализовать любой из предложенных здесь алгоритмов, вы должны суметь достаточно легко преобразовать приведенный псевдокод в код на вашем любимом языке программирования. Мы разработали псевдокод таким образом, чтобы каждый алгоритм был представлен ясно и лаконично. Вследствие этого не рассматриваются обработка ошибок и другие связанные с разработкой программного обеспечения вопросы, требующие определенных предположений, касающихся конкретной среды программирования. Авторы попытались представить каждый алгоритм просто и непосредственно, не используя индивидуальных особенностей того или иного языка программирования, что могло бы усложнить понимание сути алгоритма.

Мы понимаем, что, работая с книгой, не будучи студентом, вы не сможете проконсультироваться по поводу решения вами задач и упражнений с преподавателем. Поэтому на нашем веб-сайте `http://mitpress.mit.edu/algorithms/` имеются ссылки на решения некоторых задач и упражнений, так что у вас есть возможность проверить свою работу. Однако мы просим вас *не* присылать нам свои решения.

Коллеге

В книге приведена обширная библиография и представлены ссылки на современную литературу. В конце каждой главы есть раздел "Заключительные заме-

чания", содержащий исторические подробности и ссылки. Однако эти замечания не могут служить исчерпывающим руководством в области алгоритмов. Возможно, в это будет сложно поверить, но даже в такую объемную книгу не удалось включить многие интересные алгоритмы из-за недостатка места.

Несмотря на огромное количество писем от студентов с просьбами предоставить решения задач и упражнений, политика авторов — не приводить ссылки на источники, из которых эти задачи и упражнения были заимствованы. Это сделано для того, чтобы студенты не искали готовых решений в литературе, а решали задачи самостоятельно.

Изменения в третьем издании

Что же изменилось в третьем издании книги по сравнению со вторым? Количество этих изменений находится на том же уровне, что и количество изменений во втором издании по сравнению с первым. Как мы писали в предыдущем издании, в зависимости от точки зрения, можно сказать, что их достаточно мало или довольно много.

Беглый взгляд на оглавление книги показывает, что бóльшая часть глав и разделов второго издания осталась на месте. Мы убрали из книги две главы и один раздел, но при этом добавили три новые главы и два новых раздела, помимо этих новых глав.

Мы сохранили смешанную схему первых двух изданий. Вместо организации глав в соответствии с проблемной областью задач или в соответствии с методами решения задач, мы использовали оба подхода одновременно. В книге имеются главы, посвященные методу "разделяй и властвуй", динамическому программированию, амортизационному анализу, NP-полноте и приближенным алгоритмам. Но в ней же есть целые части, посвященные сортировке, структурам данных для динамических множеств и алгоритмам для решения задач на графах. Мы считаем, что, хотя вы и должны знать, как применять методы проектирования и анализа алгоритмов, задачи редко подсказывают, какие именно методы в наибольшей степени подходят для их решения.

Вот краткая информация о наиболее существенных изменениях в третьей редакции книги.

- Мы добавили новые главы, посвященные деревьям ван Эмде Боаса и многопоточным алгоритмам, и убрали из приложений материал о матрицах.
- Мы пересмотрели главу, посвященную рекуррентности, в сторону большего охвата метода "разделяй и властвуй" и ее первых двух разделов, посвященных применению этого метода для решения двух задач. Во втором разделе этой главы описан алгоритм Штрассена умножения матриц, которые был перенесен сюда из главы об операциях с матрицами.
- Мы убрали две главы с достаточно редко изучаемым материалом: биномиальными пирамидами и сортирующими сетями. Одна из ключевых идей главы, посвященной сортирующим сетям, а именно 0-1 принцип, в этом издании находится в задаче 8.7 в виде леммы 0-1 сортировки для алгоритмов, работающих путем сравнения и обмена. Рассмотрение пирамид Фибоначчи больше не

основывается на биномиальных пирамидах как на предшественницах пирамид Фибоначчи.

- Мы пересмотрели подход к динамическому программированию и жадным алгоритмам. Динамическое программирование теперь иллюстрируется более интересной задачей разрезания стержня, чем задача сборочного конвейера во втором издании. Кроме того, мы сделали больший, чем во втором издании, акцент на технологии запоминания и ввели понятие графа подзадачи как способа улучшения определения времени работы алгоритма динамического программирования. В нашем вводном примере для жадных алгоритмов — в задаче о выборе процессов — мы переходим к жадным алгоритмам более прямым путем, чем во втором издании.
- Способ удаления узла из бинарных деревьев поиска (включающих красно-черные деревья) теперь гарантирует, что будет удален именно тот узел, который нужно удалить. В двух первых изданиях в некоторых случаях удалялся некоторый другой узел с перемещением его содержимого в узел, переданный процедуре удаления. При таком, новом, способе удаления узлов в случае, когда другие компоненты программы поддерживают указатели на узлы дерева, они не окажутся в ситуации с устаревшими указателями на удаленные из дерева узлы.
- В материале, посвященном транспортным сетям, потоки теперь полностью базируются на ребрах, что представляет более интуитивно понятный подход, чем в первых двух изданиях.
- Поскольку материал об основах работы с матрицами и алгоритм Штрассена перенесены в другие главы, глава, посвященная матричным операциям, стала существенно меньше по сравнению со вторым изданием.
- Изменено рассмотрение алгоритма сравнения строк Кнута–Морриса–Пратта.
- Исправлен ряд ошибок. Информация о большинстве из них (но не обо всех) была получена нами через наш сайт, посвященный второму изданию книги.
- Идя навстречу многочисленным пожеланиям, мы заменили синтаксис нашего псевдокода. Теперь для указания присвоения мы используем "=", а для проверки на равенство — "==", как это делается в C, C++, Java и Python. Точно так же мы удалили ключевые слова **do** и **then** и приняли в качестве символов начала комментария до окончания строки "//". Кроме того, для указания атрибутов объектов используется запись с точкой. Но мы не зашли настолько далеко, чтобы сделать псевдокод объектно-ориентированным; он остался процедурным. Другими словами, вместо выполнения методов объектов мы просто вызываем процедуры, передавая им объекты в качестве параметров.
- Мы добавили 100 новых упражнений и 28 новых задач. Мы также обновили и расширили библиографию.
- Наконец мы прошлись по всей книге и переписали многие разделы, абзацы и отдельные предложения, делая изложение более понятным.

Веб-сайт

Вы можете воспользоваться нашим веб-сайтом по адресу `http://mitpress.mit.edu/algorithms/` для получения дополнительных материалов и для обратной связи с авторами. На сайте имеются ссылки на список известных ошибок, на решения избранных упражнений и задач и многие другие дополнительные материалы. Веб-сайт также позволяет читателю сообщить о замеченных ошибках или внести свои предложения по поводу книги.

Как создавалась эта книга

Третье издание книги, как и второе, выполнено в LaTeX 2_ε. Для математических формул мы использовали шрифт Times вместе со шрифтами MathTime Pro 2. Мы выражаем признательность за техническую поддержку Майклу Спиваку (Michael Spivak) из Publish or Perish, Inc., Лансу Карнесу (Lance Carnes) из Personal TeX, Inc., и Тиму Трегубову (Tim Tregubov) из Dartmouth College. Как и в двух предыдущих изданиях, предметный указатель компилировался с помощью Windex, написанной нами программы на языке С, а библиография — с применением BIBTeX. PDF-файлы этой книги созданы на MacBook под управлением OS 10.5.

Иллюстрации к третьему изданию созданы с использованием MacDraw Pro с соответствующими пакетами для LaTeX 2_ε для вставки в иллюстрации математических формул. К сожалению, MacDraw Pro — устаревшее программное обеспечение, не выпускающееся уже более десятилетия. Но к счастью, у нас все еще есть пара "Макинтошей", на которых можно запускать классическую среду под управлением OS 10.4, а следовательно, как правило, и MacDraw Pro. Но даже с этими сложностями мы пришли к выводу, что MacDraw Pro гораздо проще любого другого программного обеспечения для создания иллюстраций, сопровождающих текст на компьютерную тематику, и позволяет получить отличные результаты.[1] Кто знает, сколько еще протянут наши доинтеловские Маки, так что если кто-то из Apple читает эти строки, то услышьте нашу просьбу: *пожалуйста, создайте OS X-совместимую версию MacDraw Pro!*

Благодарности к третьему изданию

К настоящему времени мы сотрудничаем с MIT Press уже более двух десятилетий, и это были очень плодотворные десятилетия! Мы благодарны Эллен Фаран (Ellen Faran), Бобу Приору (Bob Prior), Аде Брунштейн (Ada Brunstein) и Мэри Рейли (Mary Reilly) за помощь и поддержку.

При написании третьего издания мы были сильно разбросаны географически, работая в Dartmouth College Department of Computer Science, MIT Computer

---

[1]Мы изучили несколько различных программ, работающих под управлением Mac OS X, но все они обладают значительными недостатками по сравнению с MacDraw Pro. Мы пробовали создавать иллюстрации для этой книги с помощью других, хорошо известных программ, но получалось, что в результате мы тратили как минимум в пять раз больше времени, чем для создания тех же иллюстраций в MacDraw Pro, причем результат оставлял желать лучшего. Так что нам пришлось скрепя сердце принять решение вернуться к MacDraw Pro на старых "Макинтошах".

Science and Artificial Intelligence Laboratory и Columbia University Department of Industrial Engineering and Operations Research. Мы благодарны нашим университетам и колледжам за создание поддерживающей и стимулирующей обстановки.

Джули Сассман (Julie Sussman, P.P.A.) вновь согласилась быть нашим техническим редактором. И вновь мы удивлялись как количеству сделанных нами ошибок, так и умению Джули их вылавливать. Она также помогла нам улучшить в ряде мест представление материала. Если бы существовал Зал славы технических редакторов, Джули, несомненно, заняла бы в нем достойное место. Она — просто феномен! Спасибо, спасибо, большое спасибо, Джули! Прия Натараджан (Priya Natarajan) также обнаружил ряд ошибок, которые мы смогли исправить еще до того, как книга была отправлена в типографию. Все оставшиеся в книге ошибки, несомненно, находятся на совести авторов (и, вероятно, были внесены в книгу после того, как ее прочла Джули).

Рассмотрение деревьев ван Эмде Боаса порождено замечаниями Эрика Демейна (Erik Demaine), на которые, в свою очередь, повлиял Майкл Бендер (Michael Bender). Мы также включили в книгу идеи Джейведа Аслама (Javed Aslam), Бредли Кусмаула (Bradley Kuszmaul) и Хью Жа (Hui Zha).

Глава, посвященная многопоточности, основана на материале, изначально написанном вместе с Гаральдом Прокопом (Harald Prokop). На этот материал большое влияние оказали некоторые другие работы в рамках проекта Cilk в MIT, включая работы Бредли Кусмаула (Bradley Kuszmaul) и Маттео Фриго (Matteo Frigo). Дизайн многопоточного псевдокода основан на расширениях MIT Cilk для языка C и расширениях Cilk Arts's Cilk++ для языка C++.

Мы также благодарим множество читателей первого и второго изданий, которые сообщали нам об ошибках или давали советы о том, как улучшить книгу. Мы исправили все найденные ошибки и приняли столько предложений, сколько смогли. Количество таких читателей столь велико, что нет никакой практической возможности перечислить их здесь.

Наконец мы выражаем благодарность нашим женам Николь Кормен (Nicole Cormen), Венди Лейзерсон (Wendy Leiserson), Гейл Ривест (Gail Rivest) и Ребекке Иври (Rebecca Ivry), а также нашим детям Рикки (Ricky), Вильяму (William), Дебби (Debby) и Кати (Katie) Лейзерсонам, Алексу (Alex) и Кристоферу (Christopher) Ривестам, а также Молли (Molly), Ною (Noah) и Бенджамену (Benjamin) Штайнам — за любовь и поддержку во время написания этой книги. Этот проект стал возможным благодаря поддержке и поощрению членов наших семей. С любовью посвящаем эту книгу им.

Томас Кормен — *Ливан, Нью-Гемпшир*
Чарльз Лейзерсон — *Кэмбридж, Массачусеттс*
Рональд Ривест — *Кэмбридж, Массачусеттс*
Клиффорд Штайн — *Нью-Йорк, Нью-Йорк*

Февраль 2009

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Адреса для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

I Основы

Введение

Эта часть книги заставит вас задуматься о вопросах, связанных с разработкой и анализом алгоритмов. Она была запланирована как вводный курс, в котором рассматриваются способы определения алгоритмов, некоторые стратегии их разработки, использующиеся в этой книге, а также применяемые в ходе анализа алгоритмов различные основополагающие идеи. Кратко рассмотрим содержание глав части I.

В главе 1 представлены обзор алгоритмов и их роль в современных вычислительных системах. В ней приводится определение алгоритма и даются некоторые примеры. Здесь также обосновывается положение о том, что алгоритмы следует рассматривать как такой же технологический продукт, как, например, аппаратное обеспечение, графические интерфейсы пользователя, объектно-ориентированные системы или сети.

В главе 2 читатель получит возможность ознакомиться с алгоритмами, с помощью которых решается задача о сортировке последовательности из n чисел. Эти алгоритмы сформулированы в виде псевдокода. Несмотря на то что используемый псевдокод напрямую не преобразуется ни в один из общепринятых языков программирования, он вполне адекватно передает структуру алгоритма, поэтому для вас не должно составлять трудности реализовать его на любом языке программирования. Для изучения выбраны алгоритм сортировки вставкой, в котором используется инкрементный подход, и алгоритм сортировки слиянием, который характеризуется применением рекурсивного метода, известного также как метод "разделяй и властвуй" (метод разбиения). В обоих алгоритмах время выполнения возрастает с увеличением количества сортируемых элементов, однако скорость этого роста зависит от выбранного алгоритма. В этой главе будет определено время работы изучаемых алгоритмов; кроме того, вы познакомитесь со специальными обозначениями для описания времени работы алгоритмов.

В главе 3 дается точное определение обозначений, введенных в главе 2 (которые называются асимптотическими обозначениями). В начале главы 3 опре-

деляется несколько асимптотических обозначений для оценки времени работы алгоритма сверху и/или снизу. Остальные разделы главы в основном посвящены математическим обозначениям. Их предназначение состоит не столько в том, чтобы ознакомить читателя с новыми математическими концепциями, сколько в том, чтобы он смог убедиться, что используемые им обозначения совпадают с принятыми в данной книге.

В главе 4 представлено дальнейшее развитие метода "разделяй и властвуй", введенного в главе 2. В главе 4 приведены дополнительные примеры алгоритмов "разделяй и властвуй", включая удивительный метод Штрассена для умножения двух квадратных матриц. В ней также представлены методы решения рекуррентных соотношений, с помощью которых описывается время работы рекурсивных алгоритмов. Одним из мощных методов является "основной метод" (master method), который используется для решения рекуррентных соотношений, возникающих в алгоритмах разбиения. Хотя немалая часть главы посвящена доказательству корректности метода контроля, вы можете его опустить, что не помешает применению метода на практике.

Глава 5 служит введением в анализ вероятностей и рандомизированные алгоритмы (т.е. алгоритмы, которые основаны на использовании случайных чисел). Анализ вероятностей обычно применяется для определения времени работы алгоритма в тех случаях, когда оно может изменяться для различных наборов входных параметров, несмотря на то что эти наборы содержат одно и то же количество параметров. В некоторых случаях можно предположить, что распределение входных величин описывается некоторым известным законом распределения вероятностей, а значит, время работы алгоритма можно усреднить по всем возможным наборам входных параметров. В других случаях распределение возникает не из-за входных значений, а в результате случайного выбора, который делается во время работы алгоритма. Алгоритм, поведение которого определяется не только входными значениями, но и величинами, полученными с помощью генератора случайных чисел, называется рандомизированным алгоритмом. Мы можем использовать рандомизированные алгоритмы для обеспечения вероятностного распределения входных данных, тем самым гарантируя, что никакой набор входных данных не приведет к низкой производительности алгоритма, или даже для ограничения числа ошибок в алгоритмах, которые могут давать ограниченное количество некорректных результатов.

В приложениях А–Г содержится дополнительный математический материал, который будет полезным в процессе чтения книги. Скорее всего, вы уже знакомы с основной частью материала, содержащегося в приложениях (хотя некоторые из встречавшихся вам ранее обозначений иногда могут отличаться от принятых в данной книге). Поэтому к приложениям следует относиться как к справочному материалу. С другой стороны, не исключено, что вы еще не знакомы с большинством вопросов, рассматриваемых в части I.

Глава 1. Роль алгоритмов в вычислениях

Что такое алгоритмы? Стоит ли тратить время на их изучение? Какова роль алгоритмов и как они соотносятся с другими компьютерными технологиями? В этой главе мы ответим на поставленные здесь вопросы.

1.1. Что такое алгоритмы

Говоря неформально, ***алгоритм*** — это любая корректно определенная вычислительная процедура, на ***вход*** (input) которой подается некоторая величина или набор величин и результатом выполнения которой является ***выходная*** (output) величина или набор значений. Таким образом, алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные.

Алгоритм также можно рассматривать как инструмент, предназначенный для решения корректно поставленной ***вычислительной задачи*** (computational problem). В постановке задачи в общих чертах задаются отношения между входом и выходом. В алгоритме описывается конкретная вычислительная процедура, с помощью которой удается добиться выполнения указанных отношений.

Например, может понадобиться выполнить сортировку последовательности чисел в неубывающем порядке. Эта задача часто возникает на практике и служит благодатной почвой для ознакомления на ее примере со многими стандартными методами разработки и анализа алгоритмов. ***Задача сортировки*** (sorting problem) формально определяется следующим образом.

Вход. Последовательность из n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход. Перестановка (переупорядочение) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности, такая, что $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Например, если на вход подается последовательность $\langle 31, 41, 59, 26, 41, 58 \rangle$, то вывод алгоритма сортировки должен быть таким: $\langle 26, 31, 41, 41, 58, 59 \rangle$. Подобная входная последовательность называется ***экземпляром*** (instance) задачи сортировки. Вообще говоря, ***экземпляр задачи*** состоит из входных данных (удовлетворяю-

щих всем ограничениям, наложенным при постановке задачи), необходимых для решения задачи.

Поскольку многие программы используют ее в качестве промежуточного шага, сортировка является основополагающей операцией в информатике, в результате чего появилось большое количество хороших алгоритмов сортировки. Выбор наиболее подходящего алгоритма зависит от многих факторов, в том числе от количества сортируемых элементов, от их порядка во входной последовательности, от возможных ограничений, накладываемых на члены последовательности, от архитектуры компьютера, а также от того, какое устройство используется для хранения последовательности: основная память, магнитные диски или даже накопители на магнитных лентах.

Говорят, что алгоритм ***корректен*** (correct), если для любых входных данных результатом его работы являются корректные выходные данные. Мы говорим, что корректный алгоритм ***решает*** данную вычислительную задачу. Если алгоритм некорректный, то для некоторых вводов он может вообще не завершить свою работу или выдать неправильный ответ. Правда, некорректные алгоритмы иногда могут оказаться полезными, если в них есть возможность контролировать частоту возникновения ошибок. Такой пример алгоритма с контролируемой частотой ошибок рассматривается в главе 31, в которой изучаются алгоритмы определения больши́х простых чисел. Тем не менее обычно мы заинтересованы только в корректных алгоритмах.

Алгоритм может быть задан на естественном языке, в виде компьютерной программы или даже воплощен в аппаратном обеспечении. Единственное требование — его спецификация должна предоставлять точное описание вычислительной процедуры, которую требуется выполнить.

Какие задачи решаются с помощью алгоритмов

Вычислительные задачи, для которых разработаны алгоритмы, отнюдь не ограничиваются сортировкой. (Возможно, об их разнообразии можно судить по объему данной книги.) Практическое применение алгоритмов чрезвычайно широко, о чем свидетельствуют приведенные ниже примеры.

- Проект по расшифровке генома человека далеко продвинулся по направлению к своей цели — к идентификации всех ста тысяч генов, входящих в состав ДНК человека, определению последовательностей, образуемых тремя миллиардами базовых пар, из которых состоит ДНК, к сортировке этой информации в базах данных и разработке инструментов для ее анализа. Для реализации всех перечисленных этапов нужны сложные алгоритмы. Хотя решение разнообразных задач, являющихся составными частями данного проекта, выходит за рамки настоящей книги, идеи, описанные во многих ее главах, используются для решения упомянутых биологических проблем. Это позволяет ученым достигать поставленных целей, эффективно используя вычислительные ресурсы. При этом экономятся время (как машинное, так и затрачиваемое сотрудниками) и деньги, а также повышается эффективность использования лабораторного оборудования.

- Интернет позволяет пользователям в любой точке мира быстро получать доступ к информации и извлекать ее в больших объемах. Благодаря помощи хитроумных алгоритмов сайты в Интернете способны работать с этими огромными объемами данных. Примерами задач, для которых жизненно необходимо применение эффективных алгоритмов, могут служить определение оптимальных маршрутов, по которым перемещаются данные (методы для решения этой задачи описываются в главе 24), и быстрый поиск страниц, на которых находится та или иная информация, с помощью специализированных поисковых машин (соответствующие методы приводятся в главах 11 и 32).
- Электронная коммерция позволяет заключать сделки и предоставлять товары и услуги с помощью различных электронных технических средств. Ее распространенность существенно зависит от способности защищать такую информацию, как номера кредитных карт, пароли и банковские счета. В число базовых технологий в этой области входят криптография с открытым ключом и цифровые подписи (они описываются в главе 31), основанные на численных алгоритмах и теории чисел.
- В производстве и коммерции очень важно распорядиться ограниченными ресурсами так, чтобы получить максимальную выгоду. Нефтяной компании может понадобиться информация о том, где пробурить скважины, чтобы получить от них как можно более высокую прибыль. Кандидат в президенты может задаться вопросом, как потратить деньги, чтобы максимально повысить свои шансы победить на выборах. Авиакомпаниям важно знать, какую минимальную цену можно назначить за билеты на тот или иной рейс, чтобы уменьшить количество свободных мест и не нарушить при этом законы, регулирующие авиаперевозку пассажиров. Провайдер Интернета должен уметь так размещать дополнительные ресурсы, чтобы повышался уровень обслуживания клиентов. Все эти задачи можно решить с помощью линейного программирования, к изучению которого мы приступим в главе 29.

Хотя некоторые детали представленных примеров и выходят за рамки настоящей книги, в ней приводятся основные методы, применяющиеся для их решения. В книге также показано, как решить многие конкретные задачи, в том числе перечисленные ниже.

- Пусть имеется карта дорог, на которой обозначены расстояния между каждой парой соседних перекрестков. Наша цель — определить кратчайший путь от одного перекрестка к другому. Количество возможных маршрутов может быть огромным, даже если исключить те из них, которые содержат самопересечения. Как найти наиболее короткий из всех возможных маршрутов? При решении этой задачи карта дорог (которая сама по себе является моделью настоящих дорог) моделируется в виде графа (мы подробнее познакомимся с графами в части VI и приложении Б). Задача будет заключаться в определении кратчайшего пути от одной вершины графа к другой. Эффективное решение этой задачи представлено в главе 24.

- У нас имеются две упорядоченные последовательности символов, $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$, и нам надо найти длиннейшую общую подпоследовательность X и Y. Подпоследовательностью X является сама последовательность X с некоторыми (возможно, всеми или никакими) удаленными элементами. Например, одной из подпоследовательностей $\langle A, B, C, D, E, F, G \rangle$ является $\langle B, C, E, G \rangle$. Наидлиннейшая общая подпоследовательность X и Y дает меру схожести двух последовательностей. Например, если этими двумя последовательностями являются базовые пары в цепочках ДНК, то мы можем считать их сходными, если они имеют длинную общую подпоследовательность. Если X имеет m символов, а Y — n символов, то X и Y имеют 2^m и 2^n возможных подпоследовательностей соответственно. Выбирая все возможные подпоследовательности X и Y и сопоставляя их, можно решить поставленную задачу только за очень длительное время (конечно, если только m и n не оказываются очень малыми величинами). В главе 15 мы познакомимся с общим методом гораздо более эффективного решения этой задачи, известным как динамическое программирование.
- У нас имеется проект, который представлен в виде библиотеки составных частей, причем каждая часть может включать экземпляры из других частей. Нам требуется перечислить все части в таком порядке, чтобы каждая часть появлялась в списке до любой другой части, ее использующей. Если проект состоит из n частей, имеется $n!$ возможных упорядочений, где $n!$ — обозначение факториала. Поскольку факториал растет быстрее даже показательной функции, мы не можем просто сгенерировать все возможные упорядочения и для каждого из них проверить, соответствует ли расположение частей нашему требованию (если, конечно, частей достаточно много). Эта задача представляет собой экземпляр топологической сортировки, и в главе 22 мы узнаем о способе ее эффективного решения.
- Пусть имеется n принадлежащих плоскости точек, и нужно найти выпуклую оболочку этих точек. Выпуклой оболочкой точек называется минимальный выпуклый многоугольник, содержащий эти точки. Для решения этой задачи удобно воспользоваться такой наглядной картиной: если представить точки в виде вбитых в доску и торчащих из нее гвоздей, то выпуклую оболочку можно получить, намотав на них резинку таким образом, чтобы все гвозди вошли внутрь замкнутой линии, образуемой резинкой. Каждый гвоздь, вокруг которого обвивается резинка, становится вершиной выпуклой оболочки (рис. 33.6 на с. 1076). В качестве набора вершин может выступать любое из 2^n подмножеств множества точек. Однако недостаточно знать, какие точки являются вершинами выпуклой оболочки, нужно еще указать порядок их обхода. Таким образом, чтобы найти выпуклую оболочку, придется перебрать множество вариантов. В главе 33 описаны два хороших метода поиска выпуклой оболочки.

Приведенные выше случаи применения алгоритмов отнюдь не являются исчерпывающими, однако на их примере выявляются две общие характеристики, присущие многим интересным алгоритмам.

1. Они имеют множество вариантов-кандидатов, подавляющее большинство которых решениями не являются. Поиск среди них работающего (или "наилучшего") кандидата является довольно сложным делом.
2. Они имеют практическое применение. Простейший пример среди перечисленных задач — определение кратчайшего пути, соединяющего два перекрестка. Любая компания, занимающаяся автомобильными или железнодорожными перевозками, финансово заинтересована в том, чтобы определить кратчайший маршрут. Это способствовало бы снижению затрат труда и расходов горючего. Маршрутизатору в Интернете также нужно иметь возможность находить кратчайшие пути в сети, чтобы как можно быстрее доставлять сообщения. Водитель, едущий из одного города в другой, может захотеть найти маршрут на веб-сайте или использовать в поездке GPS.

Не всякая задача, решаемая с помощью алгоритма, имеет просто идентифицируемое множество вариантов-кандидатов. Например, предположим, что нам дано множество числовых значений, представляющих дискретные отсчеты сигнала, и мы хотим выполнить их дискретное Фурье-преобразование. Такое дискретное Фурье-преобразование преобразует временнýю характеристику в частотную, выдавая набор числовых коэффициентов, позволяющих определить вклад различных частот в оцифрованный сигнал. Помимо того что дискретное Фурье-преобразование представляет собой основу всей обработки сигналов, оно имеет приложения в сжатии данных и умножении больших полиномов и целых чисел. В главе 30 приведены как эффективный алгоритм быстрого Фурье-преобразования (обычно именуемого БПФ (FFT)) для решения этой задачи, так и наброски проекта микросхемы для вычисления БПФ.

Структуры данных

В книге также представлен ряд структур данных. ***Структура данных*** (data structure) — это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию. Ни одна структура данных не является универсальной и не может подходить для всех целей, поэтому важно знать преимущества и ограничения, присущие некоторым из них.

Методические указания

Данную книгу можно рассматривать как "сборник рецептов" для алгоритмов. Правда, однажды вам встретится задача, для которой вы не сможете найти опубликованный алгоритм (например, таковы многие из приведенных в книге упражнений и задач). Данная книга научит вас методам разработки алгоритмов и их анализа. Это позволит вам разрабатывать корректные алгоритмы и оценивать их эффективность самостоятельно. В разных главах рассматриваются различные аспекты решения алгоритмических задач. Одни главы посвящены конкретным задачам, таким как поиск медиан и порядковых статистик в главе 9, вычисление минимальных остовных деревьев в главе 23 и определение максимального потока в сети в главе 26. Другие главы ориентированы на методы, такие как "разделяй

и властвуй" в главе 4, динамическое программирование в главе 15 и амортизационный анализ в главе 17.

Сложные задачи

Бóльшая часть этой книги посвящена эффективным алгоритмам. Обычной мерой эффективности алгоритма является скорость — время, в течение которого алгоритм выдает результат. Однако существуют задачи, для которых неизвестны эффективные методы решения. В главе 34 рассматривается интересный вопрос, имеющий отношение к подобным задачам, известным как NP-полные.

Почему NP-полные задачи представляют такой интерес? Во-первых, несмотря на то что до сих пор не найден эффективный алгоритм их решения, также не доказано, что такого алгоритма не существует. Другими словами, никто не знает, существует ли эффективный алгоритм для NP-полных задач. Во-вторых, набор NP-полных задач обладает замечательным свойством. Оно заключается в том, что если эффективный алгоритм существует хотя бы для одной из этих задач, то его можно сформулировать и для всех остальных. Эта взаимосвязь между NP-полными задачами и отсутствие методов их эффективного решения вызывают еще больший интерес к ним. В-третьих, некоторые NP-полные задачи похожи (но не идентичны) на задачи, для которых известны эффективные алгоритмы. Ученых волнует вопрос о том, как небольшое изменение формулировки задачи может значительно ухудшить эффективность самого лучшего из всех известных алгоритмов.

Вы должны знать об NP-полных задачах, поскольку в реальных приложениях некоторые из них возникают неожиданно часто. Если перед вами встанет задача найти эффективный алгоритм для NP-полной задачи, скорее всего, вы потратите много времени на безрезультатные поиски. Если же вы покажете, что данная задача принадлежит к разряду NP-полных, то можно будет вместо самого лучшего из всех возможных решений попробовать найти достаточно эффективное.

В качестве конкретного примера рассмотрим компанию грузового автотранспорта, имеющую один центральный склад. Каждый день грузовики загружаются на этом складе и отправляются по определенному маршруту, доставляя груз в несколько мест. В конце рабочего дня грузовик должен вернуться на склад, чтобы на следующее утро его снова можно было загрузить. Чтобы сократить расходы, компании нужно выбрать оптимальный порядок доставки груза в различные точки. При этом расстояние, пройденное грузовиком, должно быть минимальным. Эта задача хорошо известна как "задача о коммивояжере", и она является NP-полной. Эффективный алгоритм решения для нее неизвестен, однако при некоторых предположениях можно указать такие алгоритмы, в результате выполнения которых полученное расстояние будет ненамного превышать минимально возможное. Подобные "приближенные алгоритмы" рассматриваются в главе 35.

Параллельные вычисления

Многие годы можно было считать скорость работы процессоров устойчиво возрастающей. Однако физика накладывает свои фундаментальные ограничения

на скорость работы: поскольку с ростом тактовой частоты выделение тепловой мощности растет более чем линейно, микросхемы могут просто начать плавиться. Для повышения количества вычислений, выполняемых за единицу времени, проектировщиками все активнее используется другой путь — разработка процессоров, содержащих несколько "ядер". Мы можем уподобить эти многоядерные компьютеры нескольким компьютерам на одном обычном процессоре; другими словами, это разновидность "параллельных компьютеров". Чтобы добиться более высокой производительности от многоядерных компьютеров, мы должны разрабатывать алгоритмы с учетом возможности параллельных вычислений. В главе 27 представлена модель для "многопоточных" (multithreaded) алгоритмов, которая использует преимущества многоядерности. Эта модель обладает рядом преимуществ с теоретической точки зрения и образует основу для ряда успешных программ, включая программу для игры в шахматы.

Упражнения

1.1.1

Приведите реальные примеры задач, в которых возникает потребность в сортировке или вычислении выпуклой оболочки.

1.1.2

Какими еще параметрами, кроме скорости, можно характеризовать алгоритм на практике?

1.1.3

Выберите одну из встречавшихся вам ранее структур данных и опишите ее преимущества и ограничения.

1.1.4

Что общего между задачей об определении кратчайшего пути и задачей о коммивояжере? Чем они различаются?

1.1.5

Сформулируйте задачу, в которой необходимо только наилучшее решение. Сформулируйте также задачу, в которой может быть приемлемым решение, достаточно близкое к наилучшему.

1.2. Алгоритмы как технология

Предположим, быстродействие компьютера и объем его памяти можно увеличивать до бесконечности. Была бы в таком случае необходимость в изучении алгоритмов? Была бы, но только для того, чтобы продемонстрировать, что метод решения имеет конечное время работы и что он дает правильный ответ.

Если бы компьютеры были неограниченно быстрыми, подошел бы любой корректный метод решения задачи. Возможно, вы бы предпочли, чтобы реализация решения была выдержана в хороших традициях программирования (например, ваша реализация должна быть качественно разработана и аккуратно задокументирована), но чаще всего выбирался бы метод, который легче всего реализовать.

Конечно же, сегодня есть весьма производительные компьютеры, но их быстродействие не может быть бесконечно большим. Память также дешевеет, но не может стать бесплатной. Таким образом, время вычисления — такой же ограниченный ресурс, как и объем необходимой памяти. Вы должны разумно распоряжаться этими ресурсами, чему и способствует применение алгоритмов, эффективных в плане расходов времени и памяти.

Эффективность

Различные алгоритмы, разработанные для решения одной и той же задачи, часто очень сильно различаются по эффективности. Эти различия могут быть намного значительнее тех, которые вызваны применением неодинакового аппаратного и программного обеспечения.

В качестве примера можно привести два алгоритма сортировки, которые рассматриваются в главе 2. Для выполнения первого из них, известного как ***сортировка вставкой***, требуется время, которое оценивается как c_1n^2, где n — количество сортируемых элементов, а c_1 — константа, не зависящая от n. Таким образом, время работы этого алгоритма приблизительно пропорционально n^2. Для выполнения второго алгоритма, ***сортировки слиянием***, требуется время, приблизительно равное $c_2 n \lg n$, где $\lg n$ — краткая запись $\log_2 n$, а c_2 — некоторая другая константа, не зависящая от n. Типичная константа метода вставок меньше константы метода слияния, т.е. $c_1 < c_2$. Давайте убедимся, что постоянные множители намного меньше влияют на время работы алгоритма, чем множители, зависящие от n. Запишем время работы алгоритма сортировки вставкой как $c_1 n \cdot n$, а сортировки слиянием — как $c_2 n \cdot \lg n$. Тогда мы увидим, что там, где сортировка вставкой имеет сомножитель n, сортировка слиянием содержит $\lg n$, что существенно меньше. (Например, когда $n = 1000$, $\lg n$ приближенно равен 10, а когда n равно миллиону, $\lg n$ всего лишь около 20.) Хотя сортировка вставкой обычно работает быстрее сортировки слиянием для небольшого количества сортируемых элементов, когда размер входных данных n становится достаточно большим, все заметнее проявляется преимущество сортировки слиянием, возникающее благодаря тому, что для больших n незначительная величина $\lg n$ по сравнению с n полностью компенсирует разницу величин постоянных множителей. Не имеет значения, во сколько раз константа c_1 меньше, чем c_2. С ростом количества сортируемых элементов обязательно будет достигнут переломный момент, когда сортировка слиянием окажется более производительной.

В качестве примера рассмотрим два компьютера — А и Б. Компьютер А более быстрый, и на нем работает алгоритм сортировки вставкой, а компьютер Б более медленный, и на нем работает алгоритм сортировки методом слияния. Оба компьютера должны выполнить сортировку множества, состоящего из десяти

2 Зак. 3726

миллионов чисел. (Хотя десять миллионов чисел и могут показаться огромным количеством, если эти числа представляют собой восьмибайтовые целые числа, то входные данные занимают около 80 Мбайт памяти, что весьма немного даже для старых недорогих лэптопов.) Предположим, что компьютер А выполняет десять миллиардов команд в секунду (что быстрее любого одного последовательного компьютера на момент написания книги), а компьютер Б — только десять миллионов команд в секунду, так что компьютер А в тысячу раз быстрее компьютера Б. Чтобы различие стало еще бóльшим, предположим, что код для метода вставок (т.е. для компьютера А) написан самым лучшим в мире программистом на машинном языке и для сортировки n чисел надо выполнить $2n^2$ команд. Сортировка же методом слияния (на компьютере Б) реализована программистом-середнячком с помощью языка высокого уровня. При этом компилятор оказался не слишком эффективным, и в результате получился код, требующий выполнения $50n \lg n$ команд. Для сортировки десяти миллионов чисел компьютеру А понадобится

$$\frac{2 \cdot (10^7)^2 \text{ команд}}{10^{10} \text{ команд в секунду}} = 20\,000 \text{ секунд (более 5.5 часов)} ,$$

в то время как компьютеру Б потребуется

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ команд}}{10^7 \text{ команд в секунду}} \approx 1163 \text{ секунд (менее 20 минут)} .$$

Как видите, использование кода, время работы которого возрастает медленнее, даже при плохом компиляторе на более медленном компьютере требует более чем в 17 раз меньше процессорного времени! Если же нужно выполнить сортировку ста миллионов чисел, то преимущество метода слияния становится еще более очевидным: там, где для сортировки вставкой потребуется более 23 дней, сортировка слиянием справится за четыре часа. Общее правило таково: чем больше количество сортируемых элементов, тем заметнее преимущество сортировки слиянием.

Алгоритмы и другие технологии

Приведенный выше пример демонстрирует, что алгоритмы, как и аппаратное обеспечение компьютера, следует рассматривать как ***технологию***. Общая производительность системы настолько же зависит от эффективности алгоритма, насколько и от мощности применяющегося аппаратного обеспечения. В области разработки алгоритмов происходит такое же быстрое развитие, как и в других компьютерных технологиях.

Возникает вопрос, действительно ли так важны алгоритмы, работающие на современных компьютерах, если и так достигнуты выдающиеся успехи в других областях высоких технологий, таких как

- усовершенствованные архитектуры компьютеров и технологий их изготовления;
- легкодоступные, интуитивно понятные графические интерфейсы (GUI);

- объектно-ориентированные системы;
- интегрированные веб-технологии;
- быстрые сети, как проводные, так и беспроводные.

Ответ — да, безусловно. Несмотря на то что некоторые приложения не требуют алгоритмического наполнения явно (такие, как некоторые простые веб-приложения), для большинства приложений оно необходимо. Например, рассмотрим веб-службу, определяющую, как добраться из одного места в другое. В основе ее реализации лежит высокопроизводительное аппаратное обеспечение, графический интерфейс пользователя, глобальная сеть и, возможно, объектно-ориентированный подход. Кроме того, использование алгоритмов необходимо для определенных операций, выполняемых данной веб-службой, например таких, как поиск маршрутов (вероятно, использующий алгоритм поиска кратчайшего пути), визуализация карт и интерполяция адресов.

Более того, даже приложение, не требующее алгоритмического наполнения на высоком уровне, сильно зависит от алгоритмов. Ведь известно, что работа приложения зависит от производительности аппаратного обеспечения, а при его разработке применяются разнообразные алгоритмы. Все мы также знаем, что приложение тесно связано с графическим интерфейсом пользователя, а для разработки любого графического интерфейса пользователя требуются алгоритмы. Вспомним приложения, работающие в сети. Чтобы они могли функционировать, необходимо осуществлять маршрутизацию, которая, как уже говорилось, основана на ряде алгоритмов. Чаще всего приложения составляются на языке, отличном от машинного. Их код обрабатывается компилятором или интерпретатором, который интенсивно использует различные алгоритмы. И таким примерам несть числа.

Кроме того, ввиду постоянного роста вычислительных возможностей компьютеров, они применяются для решения все более и более сложных задач. Как мы уже убедились на примере сравнительного анализа двух методов сортировки, с ростом сложности решаемой задачи различия в эффективности алгоритмов проявляются все значительнее.

Знание основных алгоритмов и методов их разработки — одна из характеристик, отличающих действительно умелого, опытного программиста от новичка. Располагая современными компьютерными технологиями, некоторые задачи можно решить и без основательного знания алгоритмов, однако знания в этой области позволяют достичь намного большего.

Упражнения

1.2.1

Приведите пример приложения, для которого необходимо алгоритмическое наполнение на уровне приложений, и обсудите функции этих алгоритмов.

1.2.2

Предположим, на одной и той же машине проводится сравнительный анализ реализаций двух алгоритмов сортировки, работающих вставкой и слиянием. Для

сортировки n элементов вставкой необходимо $8n^2$ шагов, а для сортировки слиянием — $64n \lg n$ шагов. При каком значении n время сортировки вставкой превысит время сортировки слиянием?

1.2.3

При каком минимальном значении n алгоритм, время работы которого определяется формулой $100n^2$, работает быстрее, чем алгоритм, время работы которого выражается как 2^n, если оба алгоритма выполняются на одной и той же машине?

Задачи

1.1. Сравнение времени работы алгоритмов

Ниже приведена таблица, строки которой соответствуют различным функциям $f(n)$, а столбцы — значениям времени t. Заполните таблицу максимальными значениями n, для которых задача может быть решена за время t, если предполагается, что время работы алгоритма, необходимое для решения задачи, равно $f(n)$ микросекунд.

	Секунда	Минута	Час	День	Месяц	Год	Век
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Заключительные замечания

Имеется множество отличных учебников, посвященных общим вопросам алгоритмов. К ним относятся книги Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [5, 6][1]; Бейза (Baase) и Ван Гельдера (Van Gelder) [27]; Брассарда (Brassard) и Брейтли (Bratley) [53]; Дасгупта (Dasgupta), Пападимитроу (Papadimitriou) и Вазирани (Vazirani) [81]; Гудрича (Goodrich) и Тамазии (Tamassia) [147]; Гофри (Hofri) [174]; Горовитца (Horowitz), Сани (Sahni) и Раджасекарана (Rajasekaran) [180]; Джонсонбауфа (Johnsonbaugh) и Шефера (Schaefer) [192];

[1] Имеется русский перевод: А. Ахо, Д. Хопкрофт, Д. Ульман. *Структуры данных и алгоритмы*. — М.: И.Д. "Вильямс", 2000.

Кингстона [204]; Кляйнберга (Kleinberg) и Тардоса (Tardos) [207]; Кнута (Knuth) [208–210][2]; Козена (Kozen) [219]; Левитина (Levitin) [234]; Манбера (Manber) [241]; Мельхорна (Mehlhorn) [247–249]; Пурдома (Purdom) и Брауна (Brown) [285]; Рейнгольда (Reingold), Ньевергельта (Nievergelt) и Део (Deo) [291]; Седжевика (Sedgewick) [304]; Седжевика (Sedgewick) и Флажоле (Flajolet) [305]; Скьены (Skiena) [316]; и Вильфа (Wilf) [354]. Некоторые аспекты разработки алгоритмов, имеющие бóльшую практическую направленность, обсуждаются в книгах Бентли (Bentley) [41, 42] и Гонне (Gonnet) [144]. Обзоры по алгоритмам можно также найти в книгах *Handbook of Theoretical Computer Science, Volume A* [340] и CRC *Algorithms and Theory of Computation Handbook* [24]. Обзоры алгоритмов, применяющихся в вычислительной биологии, можно найти в учебниках Гасфилда (Gusfield) [155], Певзнера (Pevzner) [273], Сетубала (Setubal) и Мейданиса (Meidanis) [308], а также Вотермана (Waterman) [348].

---

[2]Имеется русский перевод этих книг: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000; Д. Кнут. *Искусство программирования, т. 2. Получисленные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000; Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000. Кроме того, уже после написания данной книги вышел очередной том "*Искусства программирования*": Д. Кнут. *Искусство программирования, т. 4, А. Комбинаторные алгоритмы, часть 1.* — М.: И.Д. "Вильямс", 2013.

Глава 2. Приступаем к изучению

В этой главе вы ознакомитесь с основными понятиями, с помощью которых на протяжении всей книги будет проводиться разработка и анализ алгоритмов. Глава самодостаточна, но включает ряд ссылок на материал, который будет введен в главах 3 и 4 (в ней также имеются некоторые суммы, работа с которыми рассматривается в приложении А).

В начале главы исследуется алгоритм сортировки вставками. Он предназначен для решения задачи сортировки, поставленной в главе 1. Мы определим "псевдокод", который должен быть понятен вам, если вы когда-либо занимались программированием, и применим его, чтобы показать, как будут определяться в книге наши алгоритмы. Определяя алгоритм сортировки вставкой, мы докажем его корректность и проанализируем время его работы. В ходе анализа вводятся обозначения, используемые для указания зависимости времени работы алгоритма от количества сортируемых элементов. После обсуждения сортировки вставками описывается метод декомпозиции, основанный на принципе "разделяй и властвуй". Этот подход используется для разработки различных алгоритмов; в данном случае с его помощью будет сформулирован алгоритм, называемый сортировкой слиянием. В конце главы анализируется время работы этого алгоритма сортировки.

2.1. Сортировка вставкой

Наш первый алгоритм, алгоритм сортировки вставкой, предназначен для решения ***задачи сортировки***, поставленной в главе 1.

Вход. Последовательность n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход. Перестановка (переупорядочение) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности, такая, что $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Сортируемые числа также известны под названием ***ключи*** (keys). Хотя концептуально мы сортируем последовательность, входные данные мы получаем в виде массива с n элементами.

В этой книге алгоритмы обычно описываются в виде *псевдокода*, который во многих отношениях похож на языки программирования C, C++, Java, Python или Pascal. Тем, кто знаком хотя бы с одним из этих языков, не потребуется много

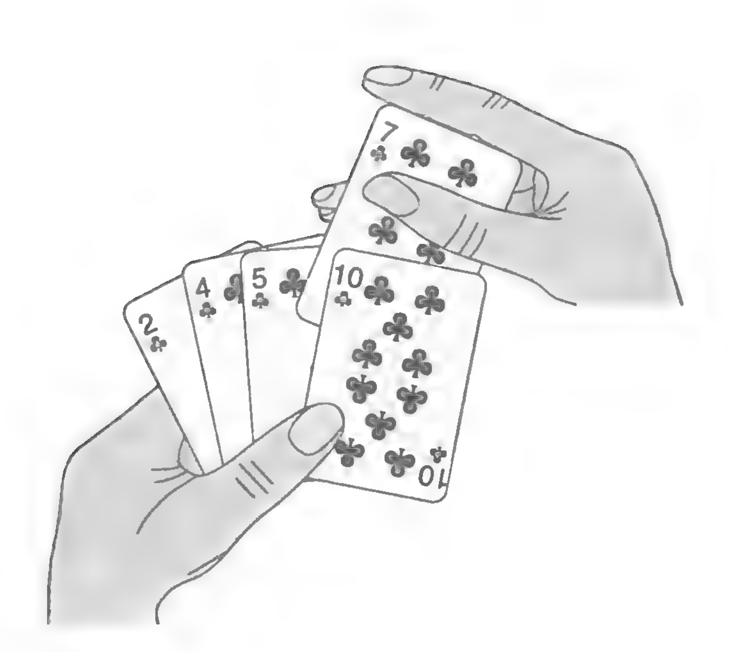


Рис. 2.1. Сортировка карт вставкой.

усилий на чтение алгоритмов. От обычного кода псевдокод отличается тем, что он выразителен, а также лаконично, точно и понятно описывает алгоритм. Иногда в качестве такового выступает литературный язык, поэтому не удивляйтесь, если в инструкции "настоящего" кода встретите обычную фразу или предложение. Другое различие между псевдокодом и обычным кодом заключается в том, что в псевдокоде, как правило, не рассматриваются некоторые вопросы, которые приходится решать разработчикам программного обеспечения. Такие вопросы, как абстракция данных, модульность и обработка ошибок, часто игнорируются, чтобы более выразительно передать суть алгоритма.

Наше изучение алгоритмов начинается с рассмотрения ***сортировки вставкой*** (insertion sort). Этот алгоритм эффективно работает при сортировке небольшого количества элементов. Сортировка вставкой напоминает способ, к которому прибегают игроки для сортировки имеющихся на руках карт. Пусть вначале в левой руке нет ни одной карты и все они лежат на столе рубашкой вверх. Далее со стола берется по одной карте, каждая из которых помещается в нужное место среди карт, которые находятся в левой руке. Чтобы определить, куда нужно поместить очередную карту, ее масть и достоинство сравниваются с мастью и достоинством карт в руке. Допустим, сравнение проводится в направлении слева направо (рис. 2.1). В любой момент карты в левой руке будут отсортированы, и это будут те карты, которые первоначально лежали в стопке на столе.

Псевдокод сортировки вставкой представлен ниже в виде процедуры под названием INSERTION-SORT, которая получает в качестве параметра массив $A[1..n]$, содержащий последовательность длиной n, которая должна быть отсортирована. (В коде количество элементов n в массиве A обозначается как $A.length$.) Алгоритм сортирует входные числа ***на месте***, без привлечения дополнительной памяти: она выполняет перестановку чисел в пределах массива A, и объем используемой при этом дополнительной памяти в любой момент работы алгоритма не превышает некоторую постоянную величину. По завершении проце-

дуры INSERTION-SORT входной массив A содержит отсортированную выходную последовательность.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Вставка A[j] в отсортированную
           последовательность A[1..j − 1].
4      i = j − 1
5      while i > 0 и A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Инварианты цикла и корректность сортировки вставкой

На рис. 2.2 показано, как этот алгоритм работает с массивом $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Индекс j указывает "текущую карту", которая помещается в руку. В начале каждой итерации цикла **for** с индексом j массив A состоит из двух частей. Элементы $A[1..j-1]$ соответствуют отсортированным картам в руке, а элементы $A[j+1..n]$ — стопке карт, которые пока что остались на столе. Заметим, что элементы $A[1..j-1]$ *изначально* также находились в позициях от 1 до $j-1$, но в другом порядке, однако теперь они отсортированы. Назовем это свойство элементов $A[1..j-1]$ ***инвариантом цикла*** (loop invariant) и сформулируем его еще раз.

> В начале каждой итерации цикла **for**, состоящего из строк 1–8, подмассив $A[1..j-1]$ состоит из элементов, которые изначально находились в $A[1..j-1]$, но теперь расположены в отсортированном порядке.

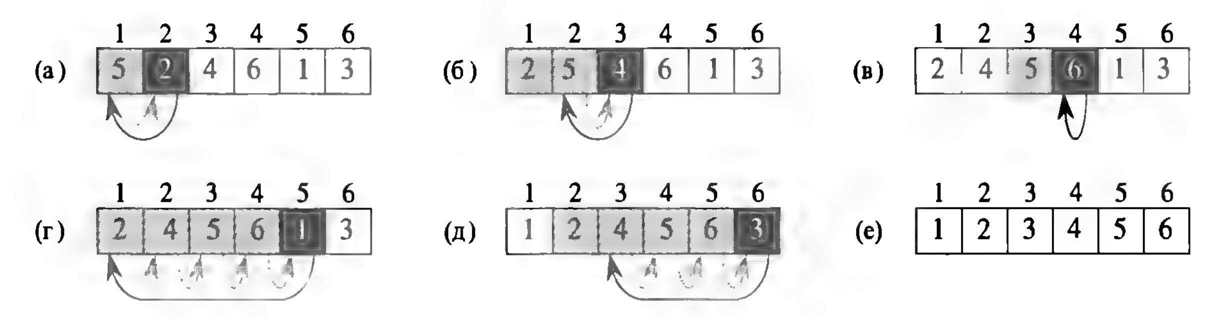


Рис. 2.2. Операции процедуры INSERTION-SORT над массивом $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Элементы массива обозначены квадратиками, над которыми находятся индексы, а внутри — значения соответствующих элементов. Части **(а)–(д)** этого рисунка соответствуют итерациям цикла **for** в строках 1–8 псевдокода. В каждой итерации черный квадратик содержит значение ключа из $A[j]$, которое сравнивается со значениями серых квадратиков, расположенных слева от него (строка псевдокода 5). Серыми стрелками указаны те значения массива, которые сдвигаются на одну позицию вправо (строка 6), а черной стрелкой — перемещение ключа (строка 8). В части **(е)** показано конечное состояние отсортированного массива.

Инварианты цикла позволяют понять, корректно ли работает алгоритм. Необходимо показать, что инварианты циклов обладают следующими тремя свойствами.

Инициализация. Они справедливы перед первой итерацией цикла.

Сохранение. Если они истинны перед очередной итерацией цикла, то остаются истинны и после нее.

Завершение. По завершении цикла инварианты позволяют убедиться в правильности алгоритма.

Если выполняются первые два свойства, инварианты цикла остаются истинными перед каждой очередной итерацией цикла. (Конечно, для доказательства того, что инвариант остается истинным перед каждой итерацией, можно использовать любые другие установленные факты, помимо самогó инварианта.) Обратите внимание на сходство с математической индукцией, когда для доказательства определенного свойства для всех элементов упорядоченной последовательности нужно доказать его справедливость для начального элемента этой последовательности, а затем обосновать шаг индукции. В данном случае первой части доказательства соответствует обоснование того, что инвариант цикла выполняется перед первой итерацией, а второй части — доказательство того, что инвариант цикла выполняется после очередной итерации (шаг индукции).

Для наших целей третье свойство, пожалуй, самое важное, так как нам нужно с помощью инварианта цикла продемонстрировать корректность алгоритма. Обычно инвариант цикла используется вместе с условием, заставляющим цикл завершиться. Свойство завершения отличает рассматриваемый нами метод от обычной математической индукции, в которой шаг индукции используется в бесконечных последовательностях. В данном случае по окончании цикла "индукция" останавливается.

Рассмотрим, соблюдаются ли эти свойства для сортировки методом вставки.

Инициализация. Начнем с того, что покажем справедливость инварианта цикла перед первой итерацией, т.е. при $j = 2$.[1] Таким образом, подмассив $A[1\,..\,j-1]$ состоит только из одного элемента $A[1]$, сохраняющего исходное значение. Более того, в этом подмножестве элементы рассортированы (тривиальное утверждение). Все вышесказанное подтверждает, что инвариант цикла соблюдается перед первой итерацией цикла.

Сохранение. Далее обоснуем второе свойство: покажем, что инвариант цикла сохраняется после каждой итерации. Выражаясь неформально, можно сказать, что в теле внешнего цикла **for** происходит сдвиг элементов $A[j-1]$, $A[j-2]$,

---

[1] Если рассматривается цикл **for**, момент времени, когда проверяется справедливость инварианта цикла перед первой итерацией, наступает сразу после начального присваивания значения индексу цикла, непосредственно перед первой проверкой в заголовочной инструкции цикла. В процедуре INSERTION-SORT это момент, когда переменной j присвоено значение 2, но еще не выполнена проверка неравенства $j \leq A.\mathit{length}$.

$A[j-3], \ldots$ на одну позицию вправо до тех пор, пока не освободится подходящее место для элемента $A[j]$ (строки 4–7), куда и вставляется значение $A[j]$ (строка 8). Подмассив $A[1\,..\,j]$ после этого состоит из элементов, изначально находившихся в $A[1\,..\,j]$, но в отсортированном порядке. Следующее за этим увеличение j для следующей итерации цикла **for** сохраняет инвариант цикла.

При более формальном подходе к рассмотрению второго свойства потребовалось бы сформулировать и обосновать инвариант для внутреннего цикла **while** в строках 5–7. Однако на данном этапе мы предпочитаем не вдаваться в такие формальные подробности, поэтому будем довольствоваться неформальным анализом, чтобы показать, что для внешнего цикла соблюдается второе свойство инварианта цикла.

Завершение. Наконец посмотрим, что происходит по завершении работы цикла. Условие, приводящее к завершению цикла **for**, — $j > A.length = n$. Поскольку каждая итерация цикла увеличивает j на 1, мы должны в этот момент иметь $j = n + 1$. Подставив в формулировку инварианта цикла вместо j значение $n+1$, получим, что подмассив $A[1\,..\,n]$ состоит из элементов, изначально находившихся в $A[1\,..\,n]$, но расположенных в отсортированном порядке. Заметим, что подмассив $A[1\,..\,n]$ и есть сам массив A, так что весь массив отсортирован, а следовательно, алгоритм корректен.

Метод инвариантов циклов будет применяться далее в данной главе, а также в последующих главах книги.

Соглашения, принятые при составлении псевдокода

При составлении псевдокода используются следующие соглашения.

- Блочная структура указывается с помощью отступов. Например, тело цикла **for**, начинающегося в строке 1, состоит из строк 2–8, а тело цикла **while**, начинающегося в строке 5, содержит строки 6 и 7, но не строку 8. Наш способ применения отступов применим и к инструкциям **if-else**[2]. Применение отступов вместо явного указания блочной структуры, такого, как с использованием ключевых слов **begin** и **end**, существенно уменьшает зашумленность кода при сохранении или даже при повышении его понятности[3].
- Конструкции циклов **while**, **for** и **repeat-until**, а также условная конструкция **if-else** интерпретируются, как в языках программирования C, C++, Java, Python

---

[2]В инструкции **if-else** мы смещаем блок **else** точно так же, как и соответствующий ему блок **if**. Хотя мы опускаем слово **then**, мы время от времени ссылаемся на часть кода, выполняемую, когда проверка **if** оказывается истинной, как на ***блок then***. В случае тестов с ветвлением мы используем ключевое слово **elseif** для проверок, следующих за первой.

[3]В этой книге каждая процедура, записанная с применением псевдокода, располагается на одной странице, так что у вас не будет проблем с угадыванием уровня отступа в коде, размещенном на нескольких страницах.

и Pascal[4]. В этой книге счетчик цикла сохраняет свое значение после выхода из цикла, в отличие от некоторых ситуаций, встречающихся в языках C++, Java и Pascal. Таким образом, непосредственно после цикла **for** значение его счетчика представляет собой значение, которое впервые превышает границу цикла **for**. Мы использовали это свойство в нашем доказательстве корректности сортировки вставкой. Заголовок цикла **for** в строке 1 имеет вид **for** $j = 2$ **to** $A.length$, так что, когда цикл завершает работу, $j = A.length+1$ (или, что то же самое, $j = n+1$, поскольку $n = A.length$). Мы используем ключевое слово **to**, когда цикл **for** увеличивает значение счетчика цикла на каждой итерации, и слово **downto**, когда на каждой итерации цикла **for** значение счетчика уменьшается. Когда значение счетчика цикла изменяется на значение, большее 1, это значение следует за необязательным ключевым словом **by**.

- Символ "//" указывает, что остальная часть строки представляет собой комментарий.
- Множественное присваивание вида $i = j = e$ присваивает обеим переменным i и j значение выражения e; его следует рассматривать как эквивалентное присваиванию $j = e$, за которым следует присваивание $i = j$.
- Переменные (такие, как i, j или key) являются локальными для данной процедуры. Мы не будем использовать глобальные переменные без явного указания этого факта.
- Доступ к элементам массива осуществляется путем указания имени массива, за которым в квадратных скобках следует индекс. Например, $A[i]$ указывает i-й элемент массива A. Запись ".." используется для того, чтобы указать диапазон значений индексов массива. Так, $A[1..j]$ определяет подмассив массива A, состоящий из j элементов $A[1], A[2], \ldots, A[j]$.
- Составные данные мы обычно организуем в ***объекты***, которые состоят из ***атрибутов***. К определенному атрибуту объекта мы обращаемся с применением синтаксиса, имеющегося во многих объектно-ориентированных языках программирования: за именем объекта следует точка, за которой следует имя атрибута. Например, мы рассматриваем массив как объект с атрибутом $length$, указывающим количество содержащихся в нем элементов. Чтобы указать количество элементов в массиве A, мы записываем $A.length$.

  Мы рассматриваем переменную, представляющую массив или объект, как указатель на данные, представляющие этот массив или объект. Для всех атрибутов f объекта x присваивание $y = x$ приводит к тому, что $y.f$ становится равным $x.f$. Более того, если мы теперь установим $x.f = 3$, то после этого не только $x.f$ будет равным 3, но и $y.f$ также станет равным 3. Другими словами, x и y после присваивания $y = x$ указывают на один и тот же объект.

---

[4]Большинство языков с блочной структурой имеют эквивалентные конструкции, хотя точные их синтаксисы могут различаться. В языке Python отсутствуют циклы **repeat-until**, а его циклы **for** работают немного не так, как циклы **for** в этой книге.

Запись атрибутов может быть "каскадной". Например, предположим, что атрибут f сам по себе является указателем на объект некоторого типа, который имеет атрибут g. Тогда запись $x.f.g$ неявно подразумевает наличие скобок $(x.f).g$. Другими словами, если мы присвоим $y = x.f$, то $x.f.g$ будет тем же, что и $y.g$.

Иногда указатель вообще не ссылается ни на какой объект. В таком случае он имеет специальное значение NIL.

- Параметры в процедуру передаются ***по значению***: вызываемая процедура получает собственную копию параметров, и если она присваивает параметру значение, то это изменение *не* будет видимым для вызывающей процедуры. При передаче объектов копируется указатель на представляющие объект данные, но не атрибуты объекта. Например, если x представляет собой параметр вызываемой процедуры, присваивание $x = y$ в вызываемой процедуре не будет видимым для вызывающей процедуры. Однако присваивание $x.f = 3$ является видимым. Аналогично массивы передаются с помощью передачи указателя, а не целого массива, и изменения отдельных элементов массива видимы для вызывающей процедуры.
- Инструкция **return** немедленно возвращает управление в точку вызова в вызывающей процедуре. В большинстве случаев инструкция **return** получает значение для возврата вызывающей процедуре. Наш псевдокод отличается от многих языков программирования тем, что мы допускаем возврат нескольких значений одной инструкцией **return**.
- Булевы операторы "и" и "или" вычисляются ***сокращенно*** (short circuiting). Это означает, что при вычислении выражения "x и y" сначала вычисляется значение выражения x. Если это значение ложно (FALSE), то все выражение не может быть истинным, и значение выражения y не вычисляется. Если же выражение x истинно (TRUE), то для определения значения всего выражения необходимо вычислить выражение y. Аналогично в выражении "x или y" величина y вычисляется только в том случае, если выражение x ложно. Сокращенные операторы позволяют составлять такие логические выражения, как "$x \neq$ NIL и $x.f = y$", не беспокоясь о том, что произойдет при попытке вычислить выражение $x.f$, когда x равно NIL.
- Ключевое слово **error** указывает на ошибку, произошедшую из-за неверных условий при вызове процедуры. За обработку ошибки отвечает вызывающая процедура, а потому мы не указываем, какие действия должны быть предприняты.

Упражнения

2.1.1

Используя рис. 2.2 в качестве образца, проиллюстрируйте работу процедуры INSERTION-SORT по сортировке массива $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1.2

Перепишите процедуру INSERTION-SORT для сортировки в невозрастающем порядке вместо неубывающего.

2.1.3

Рассмотрим ***задачу поиска***.

Вход. Последовательность из n чисел $A = \langle a_1, a_2, \ldots, a_n \rangle$ и значение v.

Выход. Индекс i, такой, что $v = A[i]$, или специальное значение NIL, если v в A отсутствует.

Составьте псевдокод ***линейного поиска***, при работе которого выполняется сканирование последовательности в поисках значения v. Докажите корректность алгоритма с помощью инварианта цикла. Убедитесь, что выбранный инвариант цикла удовлетворяет трем необходимым условиям.

2.1.4

Рассмотрим задачу сложения двух n-битовых двоичных целых чисел, хранящихся в n-элементных массивах A и B. Сумму этих двух чисел необходимо занести в двоичной форме в $(n + 1)$-элементный массив C. Приведите строгую формулировку задачи и составьте псевдокод для сложения этих двух чисел.

2.2. Анализ алгоритмов

Анализ заключается в том, чтобы предсказать требуемые для его выполнения ресурсы. Иногда оценивается потребность в таких ресурсах, как память, пропускная способность сети или необходимое аппаратное обеспечение, но чаще всего определяется время вычисления. Путем анализа нескольких алгоритмов, предназначенных для решения одной и той же задачи, можно выбрать наиболее эффективный из них. В процессе такого анализа может также оказаться, что несколько алгоритмов примерно равноценны, а многие алгоритмы в процессе анализа часто приходится отбрасывать.

Прежде чем мы научимся анализировать алгоритмы, необходимо разработать технологию, которая будет для этого использоваться. В эту технологию нужно будет включить модель ресурсов и величины их измерения. С учетом того, что алгоритмы реализуются в виде компьютерных программ, в этой книге в большинстве случаев в качестве технологии реализации принята модель обобщенной однопроцессорной машины с ***памятью с произвольным доступом*** (Random-Access Machine — RAM). В этой модели команды процессора выполняются последовательно; одновременно выполняемые операции отсутствуют.

Строго говоря, в модели RAM следует точно определить набор инструкций и время их выполнения, однако это утомительно и мало способствует пониманию принципов разработки и анализа алгоритмов. С другой стороны, нужно соблюдать осторожность, чтобы не исказить модель RAM. Например, что будет, если в RAM встроена команда сортировки? В этом случае сортировку можно выпол-

нять с помощью всего одной команды процессора. Такая модель нереалистична, поскольку настоящие компьютеры не имеют подобных встроенных команд, а мы ориентируемся именно на их устройство. В рассматриваемую модель входят те команды, которые обычно можно найти в реальных компьютерах: арифметические (сложение, вычитание, умножение, деление, вычисление остатка от деления, приближение действительного числа ближайшим меньшим или ближайшим большим целым), операции перемещения данных (загрузка, занесение в память, копирование) и управляющие (условное и безусловное ветвление, вызов подпрограммы и возврат из нее). Для выполнения каждой такой инструкции требуется определенный фиксированный промежуток времени.

В модели RAM есть целочисленный тип данных и тип чисел с плавающей точкой (для хранения действительных чисел). Несмотря на то что обычно в этой книге точность не рассматривается, в некоторых приложениях она играет важную роль. Также предполагается, что существует верхний предел размера слова данных. Например, если обрабатываются входные данные с максимальным значением n, обычно предполагается, что целые числа представлены $c \lg n$ битами для некоторой константы $c \geq 1$. Требование $c \geq 1$ обусловлено тем, что в каждом слове должно храниться одно из n значений, что позволит индексировать входные элементы. Кроме того, предполагается, что c — это конечная константа, поэтому объем слова не может увеличиваться до бесконечности. (Если бы это было возможно, в одном слове можно было бы хранить данные огромных объемов и осуществлять над ними операции в рамках одной элементарной команды, что нереально.)

В реальных компьютерах содержатся команды, не упомянутые выше, которые представляют "серую область" модели RAM. Например, является ли возведение в степень командой с константным временем работы? В общем случае — нет; чтобы вычислить выражение x^y, в котором x и y — действительные числа, потребуется несколько команд. Однако в некоторых случаях эту операцию можно представить в виде элементарной команды. Во многих компьютерах имеется команда побитового сдвига влево, которая в течение времени, требуемого для выполнения элементарной команды, сдвигает биты целого числа на k позиций влево. В большинстве случаев такой сдвиг целого числа на одну позицию эквивалентен его умножению на 2. Сдвиг битов на k позиций влево эквивалентен его умножению на 2^k. Таким образом, на этих компьютерах 2^k можно вычислить с помощью одной элементарной инструкции, сдвинув целое число 1 на k позиций влево; при этом k не должно превышать количество битов компьютерного слова. Мы попытаемся избегать таких "серых областей" модели RAM, однако вычисление 2^k будет рассматриваться как элементарная операция, если k — достаточно малое целое положительное число.

В исследуемой модели RAM не предпринимаются попытки смоделировать иерархию запоминающих устройств, общепринятую на современных компьютерах. Таким образом, мы не моделируем кеш и виртуальную память. В некоторых вычислительных моделях предпринимается попытка смоделировать эффекты, вызванные иерархией запоминающих устройств, которые иногда важны в реальных программах, работающих на реальных машинах. В ряде рассмотренных в данной

книге задач эти эффекты принимаются во внимание, но в большинстве случаев они не учитываются. Модели, включающие в себя иерархию запоминающих устройств, заметно сложнее модели RAM и поэтому могут затруднить работу. Кроме того, анализ, основанный на модели RAM, обычно замечательно предсказывает производительность алгоритмов, выполняющихся на реальных машинах.

Анализ даже простого алгоритма в модели RAM может потребовать значительных усилий. В число необходимых математических инструментов могут войти комбинаторика, теория вероятностей, навыки алгебраических преобразований и способность идентифицировать наиболее важные слагаемые в формуле. Поскольку поведение алгоритма может различаться для разных наборов входных значений, потребуется методика учета, описывающая поведение алгоритма с помощью простых и понятных формул.

Даже когда для анализа данного алгоритма выбирается всего одна модель машины, нам все еще предстоит выбрать средства для выражения анализа. Хотелось бы выбрать простые обозначения, которые позволят легко с ними работать и выявлять важные характеристики требований, предъявляемых алгоритмом к ресурсам, а также избегать сложных деталей.

Анализ сортировки вставкой

Время работы процедуры INSERTION-SORT зависит от набора входных значений: для сортировки тысячи чисел требуется больше времени, чем для сортировки трех чисел. Кроме того, время сортировки с помощью этой процедуры может быть разным для последовательностей, состоящих из одного и того же количества элементов, в зависимости от степени упорядоченности этих последовательностей до начала сортировки. В общем случае время работы алгоритма увеличивается с увеличением количества входных данных, поэтому общепринятая практика — представлять время работы программы как функцию, зависящую от количества входных элементов. Для этого понятия "время работы алгоритма" и "размер входных данных" нужно определить точнее.

Наиболее адекватное понятие ***размера входных данных*** (input size) зависит от рассматриваемой задачи. Во многих задачах, таких как сортировка или дискретные преобразования Фурье, это *количество входных элементов*, например размер n сортируемого массива. Для многих других задач, таких как перемножение двух целых чисел, наиболее подходящая мера для измерения размера ввода — *общее количество битов*, необходимых для представления входных данных в обычных двоичных обозначениях. Иногда размер ввода удобнее описывать с помощью не одного, а двух чисел. Например, если на вход алгоритма подается граф, размер ввода можно описывать, указывая количество вершин и ребер графа. Для каждой рассматриваемой далее задачи будет указываться способ измерения размера входных данных.

Время работы алгоритма для тех или иных входных данных измеряется в количестве элементарных операций, или "шагов", которые необходимо выполнить. Здесь удобно ввести понятие шага, чтобы рассуждения были как можно более машинно-независимыми. На данном этапе мы будем исходить из точки зрения, согласно которой для выполнения каждой строки псевдокода требуется фикси-

рованное время. Время выполнения различных строк может различаться, но мы предположим, что одна и та же i-я строка выполняется за время c_i, где c_i — константа. Эта точка зрения согласуется с моделью RAM и отражает особенности практической реализации псевдокода на реальных компьютерах[5].

В последующем рассмотрении формула для выражения времени работы алгоритма INSERTION-SORT, которая сначала будет сложным образом зависеть от всех величин c_i, значительно упростится благодаря более лаконичным обозначениям, с которыми проще работать. Эти более простые обозначения позволят легче определить, какой из двух алгоритмов эффективнее.

Начнем с того, что введем для процедуры INSERTION-SORT временнýю "стоимость" каждой инструкции и количество их повторений. Для каждого $j = 2, 3, \ldots, n$, где $n = A.length$, обозначим через t_j количество проверок условия в цикле **while** (строка 5). При нормальном завершении циклов **for** и **while** (т.е. когда перестает выполняться условие, заданное в заголовке цикла) условие проверяется на один раз больше, чем выполняется тело цикла. Само собой разумеется, мы считаем, что комментарии не являются исполняемыми инструкциями, поэтому они не увеличивают время работы алгоритма.

INSERTION-SORT(A)	*Стоимость*	*Повторы*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad$ // Вставка $A[j]$ в отсортированную последовательность $A[1..j-1]$.	0	$n-1$
4 $\quad i = j-1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ и $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $\qquad i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

Время работы алгоритма — это сумма промежутков времени, необходимых для выполнения каждой входящей в его состав выполняемой инструкции. Если выполнение инструкции длится в течение времени c_i и она повторяется в алгоритме n раз, то ее вклад в полное время работы алгоритма равен $c_i n$[6]. Чтобы вычислить время работы алгоритма INSERTION-SORT (обозначим его через $T(n)$), нужно просуммировать произведения значений, стоящих в столбцах *стоимость*

---

[5] Здесь есть некоторые нюансы. Шаги вычислений, описанные на обычном языке, часто представляют собой разновидности процедур, состоящих из нескольких элементарных инструкций, имеющих более чем константное время работы. Например, далее в этой книге может встретиться строка "сортировка точек по координате x". Как вы увидите, эта команда требует больше чем постоянного количества времени работы. Заметим также, что команда вызова подпрограммы выполняется в течение фиксированного времени, однако сколько длится выполнение вызванной подпрограммы, зависит от ее сложности. Таким образом, процесс вызова подпрограммы (передачу в нее параметров и другие действия) следует отличать от процесса ***выполнения*** этой подпрограммы.

[6] Это правило не всегда применимо к такому ресурсу, как память. Если инструкция оперирует m словами памяти и выполняется n раз, то это еще не означает, что всего при этом потребляется mn слов памяти.

и *повторы*, в результате чего получим

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\ .$$

Даже если размер входных данных является фиксированной величиной, время работы алгоритма может зависеть от *самих* входных данных — степени упорядоченности сортируемых величин, которой они обладали до ввода. Например, самый благоприятный случай для алгоритма INSERTION-SORT — это когда все элементы массива уже отсортированы. Тогда для каждого $j = 2, 3, \ldots, n$ мы находим, что $A[i] \leq key$ в строке 5, еще когда i равно своему начальному значению $j-1$. Таким образом, $t_j = 1$ для $j = 2, 3, \ldots, n$, и время работы алгоритма в самом благоприятном случае вычисляется так:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

Это время работы можно записать как $an + b$, где a и b — *константы*, зависящие от величин c_i; т.е. это время является ***линейной функцией*** от n.

Если же массив находится в порядке, обратном требуемому (т.е. в порядке убывания элементов), то реализуется наихудший случай. При этом мы должны сравнивать каждый элемент $A[j]$ с каждым элементом всего отсортированного подмассива $A[1\,..\,j-1]$, так что $t_j = j$ для $j = 2, 3, \ldots, n$. Заметив, что

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

и

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(см. обзор, посвященный таким суммам, в приложении А), мы находим, что в наихудшем случае время работы INSERTION-SORT составляет

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

Это время работы в наихудшем случае можно записать как $an^2 + bn + c$, где a, b и c — константы, зависящие от стоимостей c_i; таким образом, мы имеем ***квадратичную функцию*** от n.

Обычно время работы алгоритма для определенных входных данных фиксировано, как в случае сортировки вставкой, однако в последующих главах мы ознакомимся с некоторыми интересными "рандомизированными" алгоритмами, поведение которых носит вероятностный характер. Время их работы может меняться даже для одних и тех же входных данных.

Наихудшее и среднее время работы

Анализируя алгоритм сортировки вставкой, мы рассматривали как наилучший, так и наихудший случай, когда элементы массива были рассортированы в порядке, обратном требуемому. Далее в этой книге мы будем уделять основное внимание определению только времени работы в ***наихудшем случае***, т.е. максимального времени работы для *любых* входных данных размером n. На то есть три причины.

- Время работы алгоритма в наихудшем случае — это верхний предел этой величины для любых входных данных. Располагая этим значением, мы точно знаем, что для выполнения алгоритма не потребуется большее количество времени. Не нужно будет делать каких-то сложных предположений о времени работы и надеяться, что на самом деле эта величина не будет превышена.
- В некоторых алгоритмах наихудший случай встречается достаточно часто. Например, если в базе данных происходит поиск информации, то наихудшему случаю соответствует ситуация, когда нужная информация в базе данных отсутствует. В некоторых приложениях поиск отсутствующей информации может происходить довольно часто.
- Характер поведения "усредненного" времени работы часто ничем не лучше поведения времени работы для наихудшего случая. Предположим, что последовательность, к которой применяется сортировка методом вставок, сформирована случайным образом. Сколько времени понадобится, чтобы определить, в какое место подмассива $A[1..j-1]$ следует поместить элемент $A[j]$? В среднем половина элементов подмассива $A[1..j-1]$ меньше, чем $A[j]$, а половина — больше этого значения. Таким образом, в среднем нужно проверить половину элементов подмассива $A[1..j-1]$, поэтому t_j приблизительно равно $j/2$. В результате получается, что среднее время работы алгоритма является квадратичной функцией от количества входных элементов, т.е. характер этой зависимости такой же, как и для времени работы в наихудшем случае.

В некоторых частных случаях нас будет интересовать ***среднее*** время работы алгоритма, или его ***математическое ожидание***[7]. Позже будет продемонстриро-

---

[7]Далее в книге строгий термин "математическое ожидание" некоторой величины зачастую для простоты изложения заменяется термином "ожидаемое значение", например "ожидаемое время работы алгоритма" означает "математическое ожидание времени работы алгоритма". — *Примеч. ред.*

ван метод ***вероятностного анализа***, применяемый в книге к ряду алгоритмов. Однако область его использования ограничена, поскольку не всегда очевидно, какие входные данные для данной задачи являются "усредненными". Часто делается предположение, что все наборы входных параметров одного и того же объема встречаются с одинаковой вероятностью. На практике это предположение может не соблюдаться, однако иногда можно применять ***рандомизированные алгоритмы***, в которых используется случайный выбор, и это позволяет провести вероятностный анализ ***ожидаемого*** времени работы алгоритма. Более детально мы рассмотрим рандомизированные алгоритмы в главе 5 и некоторых последующих главах.

Порядок роста

Для облегчения анализа процедуры INSERTION-SORT были сделаны некоторые упрощающие предположения. Вначале мы проигнорировали фактическую стоимость выполнения каждой инструкции, представив эту величину в виде некоторой константы c_i. Далее мы увидели, что учет всех этих констант дает излишнюю информацию: время работы алгоритма в наихудшем случае выражается формулой $an^2 + bn + c$, где a, b и c — некоторые константы, зависящие от стоимостей c_i. Таким образом, мы игнорируем не только фактические стоимости команд, но и их абстрактные стоимости c_i.

Теперь введем еще одно абстрактное понятие, упрощающее анализ. Это ***скорость роста*** (rate of growth), или ***порядок роста*** (order of growth), интересующего нас времени работы. Таким образом, во внимание будет приниматься только главный член формулы (т.е. в нашем случае an^2), поскольку при больших значениях n членами меньшего порядка можно пренебречь. Кроме того, постоянный множитель при главном члене также будет игнорироваться, так как для оценки вычислительной эффективности алгоритма с входными данными большого объема они менее важны, чем порядок роста. В случае сортировки вставкой, когда мы игнорируем члены более низкого порядка и коэффициент при старшем члене, мы остаемся с единственным множителем n^2 из старшего члена. Таким образом, время работы алгоритма, работающего по методу вставок, в наихудшем случае равно $\Theta(n^2)$ (произносится "тета от n в квадрате"). В этой главе Θ-обозначения используются неформально; строго мы определим их в главе 3.

Обычно один алгоритм рассматривается как более эффективный по сравнению с другим, если время его работы в наихудшем случае имеет более низкий порядок роста. Из-за константных множителей и членов более низкого порядка алгоритм с более высоким порядком роста может выполняться для небольших входных данных быстрее, чем алгоритм с более низким порядком роста. Но для достаточно больших входных данных алгоритм $\Theta(n^2)$, например, будет в наихудшем случае работать быстрее алгоритма с $\Theta(n^3)$.

Упражнения

2.2.1

Выразите функцию $n^3/1000 - 100n^2 - 100n + 3$ в Θ-обозначениях.

2.2.2

Рассмотрим сортировку элементов массива A, которая выполняется следующим образом. Сначала определяется наименьший элемент массива A, который ставится на место элемента $A[1]$. Затем производится поиск второго наименьшего элемента массива A, который ставится на место элемента $A[2]$. Этот процесс продолжается для первых $n - 1$ элементов массива A. Запишите псевдокод этого алгоритма, известного как ***сортировка выбором*** (selection sort). Какой инвариант цикла сохраняется для этого алгоритма? Почему его достаточно выполнить для первых $n - 1$ элементов, а не для всех n элементов? Определите время работы алгоритма в наилучшем и наихудшем случаях и запишите его в Θ-обозначениях.

2.2.3

Вновь обратимся к алгоритму линейного поиска (см. упр. 2.1.3). Для скольких элементов входной последовательности в среднем нужно произвести проверку, если предполагается, что все элементы массива с равной вероятностью могут иметь искомое значение? Что происходит в наихудшем случае? Чему равно время работы алгоритма линейного поиска в среднем и в наихудшем случаях в Θ-обозначениях? Обоснуйте свой ответ.

2.2.4

Каким образом можно модифицировать почти каждый алгоритм, чтобы получить оптимальное время работы в наилучшем случае?

2.3. Разработка алгоритмов

У нас имеется богатый набор методов разработки алгоритмов. В случае сортировки вставкой мы использовали ***инкрементный*** подход: имея отсортированный подмассив $A[1..j-1]$, мы вставляем один элемент $A[j]$ в соответствующее место, получая отсортированный подмассив $A[1..j]$.

В этом разделе мы рассмотрим альтернативный подход, известный как метод декомпозиции, или метод “разделяй и властвуй”, который мы детально изучим в главе 4. Мы используем подход “разделяй и властвуй” для разработки алгоритма сортировки, время работы которого в наихудшем случае оказывается гораздо меньшим, чем время работы сортировки вставкой. Одно из преимуществ алгоритмов “разделяй и властвуй” заключается в том, что зачастую оказывается очень легко определить время работы такого алгоритма с применением методики, которая будет описана в главе 4.

2.3.1. Метод декомпозиции

Многие полезные алгоритмы имеют ***рекурсивную*** структуру: для решения поставленной задачи они рекурсивно вызывают сами себя один или несколько раз, решая вспомогательные подзадачи, тесно связанные с основной задачей. Такие алгоритмы зачастую разрабатываются с помощью метода ***декомпозиции***, или метода ***“разделяй и властвуй”***: сложная задача разбивается на несколько простых,

которые подобны исходной задаче, но имеют меньший объем; далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются для получения решения исходной задачи.

Парадигма, лежащая в основе метода декомпозиции "разделяй и властвуй", на каждом уровне рекурсии включает в себя три шага.

Разделение задачи на несколько подзадач, которые представляют собой меньшие экземпляры той же задачи.

Властвование над подзадачами путем их рекурсивного решения. Если размеры подзадач достаточно малы, такие подзадачи могут решаться непосредственно.

Комбинирование решений подзадач в решение исходной задачи.

Алгоритм ***сортировки слиянием*** (merge sort) точно следует парадигме "разделяй и властвуй". Интуитивно он работает следующим образом.

Разделение. Делим n-элементную сортируемую последовательность на две подпоследовательности по $n/2$ элементов.

Властвование. Рекурсивно сортируем эти две подпоследовательности с использованием сортировки слиянием.

Комбинирование. Соединяем две отсортированные подпоследовательности для получения окончательного отсортированного ответа.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любая такая последовательность уже является упорядоченной.

Ключевая операция, которая производится в процессе сортировки по методу слияний, — это объединение двух отсортированных последовательностей в ходе комбинирования (последний этап). Это делается с помощью вызова вспомогательной процедуры procedure MERGE(A, p, q, r), где A — массив, а p, q и r — индексы, нумерующие элементы массива, такие, что $p \leq q < r$. В этой процедуре предполагается, что элементы подмассивов $A[p\,..\,q]$ и $A[q+1\,..\,r]$ упорядочены. Она ***сливает*** эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива $A[p\,..\,r]$.

Для выполнения процедуры MERGE требуется время $\Theta(n)$, где $n = r - p + 1$ — общее количество подлежащих слиянию элементов. Процедура работает следующим образом. Возвращаясь к наглядному примеру сортировки карт, предположим, что на столе лежат две стопки карт, обращенных лицевой стороной вниз. Карты в каждой стопке отсортированы, причем наверху находится карта наименьшего достоинства. Эти две стопки нужно объединить в одну выходную, в которой карты будут рассортированы и также будут обращены рубашкой вверх. Основной шаг состоит в том, чтобы из двух младших карт выбрать самую младшую, извлечь ее из соответствующей стопки (при этом в данной стопке верхней окажется новая карта) и поместить в выходную стопку. Этот шаг повторяется до тех пор, пока в одной из входных стопок не закончатся карты, после чего оставшиеся в другой стопке карты нужно поместить в выходную стопку. С вычислительной

точки зрения выполнение каждого основного шага занимает одинаковые промежутки времени, так как все сводится к сравнению достоинства двух верхних карт. Поскольку необходимо выполнить по крайней мере n основных шагов, время работы процедуры слияния равно $\Theta(n)$.

Описанная идея реализована в представленном ниже псевдокоде, однако в нем также есть дополнительное ухищрение, благодаря которому в ходе каждого основного шага не приходится проверять, является ли каждая из двух стопок пустой. Мы помещаем в самый низ обеих объединяемых колод так называемую ***сигнальную*** карту-ограничитель особого достоинства, что позволяет упростить код. Здесь в качестве сигнального значения используется ∞, так что, когда мы встречаем карту достоинством ∞, меньшей карты мы не встретим до полного исчерпания обеих стопок. Как только мы встречаем такую карту, это означает, что все несигнальные карты уже помещены в выходную стопку. Поскольку заранее известно, что в выходной стопке должна содержаться ровно $r - p + 1$ карта, выполнив соответствующее количество основных шагов, можно остановиться.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  Пусть L[1..n1 + 1] и R[1..n2 + 1] — новые массивы
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Подробно рассмотрим работу процедуры MERGE. В строке 1 вычисляется длина n_1 подмассива $A[p..q]$, а в строке 2 вычисляется длина n_2 подмассива $A[q+1..r]$. Мы создаем массивы L и R ("левый" и "правый") с длинами $n_1 + 1$ и $n_2 + 1$ соответственно в строке 3; дополнительная ячейка в каждом массиве будут содержать ограничитель. Цикл **for** в строках 4 и 5 копирует подмассив $A[p..q]$ в $L[1..n_1]$, а цикл **for** в строках 6 и 7 копирует подмассив $A[q+1..r]$ в $R[1..n_2]$. Строки 8 и 9 помещают ограничители в последние ячейки L и R. Строки 10–17, проиллюстрированные на рис. 2.3, выполняют $r - p + 1$ базовый шаг, сохраняющий инвариант цикла.

В начале каждой итерации цикла **for** в строках 12–17 подмассив $A[p..k-1]$ содержит $k - p$ наименьших элементов $L[1..n_1 + 1]$

и $R[1..n_2+1]$ в отсортированном порядке. Кроме того, $L[i]$ и $R[j]$ являются наименьшими элементами соответствующих массивов, которые еще не были скопированы назад в A.

Необходимо показать, что этот инвариант цикла соблюдается перед первой итерацией рассматриваемого цикла **for** в строках 12–17, что каждая итерация цикла его сохраняет и что с его помощью можно продемонстрировать корректность алгоритма, когда цикл заканчивает свою работу.

Инициализация. Перед первой итерацией цикла $k = p$, так что подмассив $A[p..k-1]$ пуст. Он содержит $k - p = 0$ наименьших элементов массивов L и R, а поскольку $i = j = 1$, элементы $L[i]$ и $R[j]$ — наименьшие элементы массивов L и R, не скопированные обратно в массив A.

Сохранение. Чтобы убедиться, что инвариант цикла сохраняется после каждой итерации, сначала предположим, что $L[i] \leq R[j]$. Тогда $L[i]$ — наименьший элемент, пока еще не скопированный в массив A. Поскольку в подмассиве $A[p..k-1]$ содержится $k-p$ наименьших элементов, после копирования в строке 14 $L[i]$ в $A[k]$ в подмассиве $A[p..k]$ будет содержаться $k-p+1$ наименьших элементов. В результате увеличения параметра k цикла **for** и значения переменной i (строка 15), инвариант цикла восстанавливается перед следующей итерацией. Если же выполняется неравенство $L[i] > R[j]$, то в строках 16 и 17 выполняются соответствующие действия, в ходе которых также сохраняется инвариант цикла.

Завершение. Алгоритм завершается, когда $k = r + 1$. В соответствии с инвариантом цикла подмассив $A[p..k-1]$ (т.е. подмассив $A[p..r]$) содержит $k-p = r-p+1$ наименьших элементов массивов $L[1..n_1+1]$ и $R[1..n_2+1]$ в отсортированном порядке. Суммарное количество элементов в массивах L и R равно $n_1 + n_2 + 2 = r - p + 3$. Все они, кроме двух самых больших, скопированы обратно в массив A, а два оставшихся элемента являются сигнальными.

Чтобы убедиться, что время работы процедуры MERGE равно $\Theta(n)$, где $n = r - p + 1$, заметим, что каждая из строк 1–3 и 8–11 выполняется за константное время; длительность циклов **for** в строках 4–7 равна $\Theta(n_1 + n_2) = \Theta(n)$,[8] а в цикле **for** в строках 12–17 выполняются n итераций, на каждую из которых затрачивается константное время.

Теперь процедуру MERGE можно использовать в качестве подпрограммы в алгоритме сортировки слиянием. Процедура MERGE-SORT(A, p, r) выполняет сортировку элементов в подмассиве $A[p..r]$. Если справедливо неравенство $p \geq r$, то в этом подмассиве содержится не более одного элемента, и, таким образом, он является уже отсортированным. В противном случае производится разбиение,

---

[8] В главе 3 будет показано, как формально интерпретируются уравнения с Θ-обозначениями.

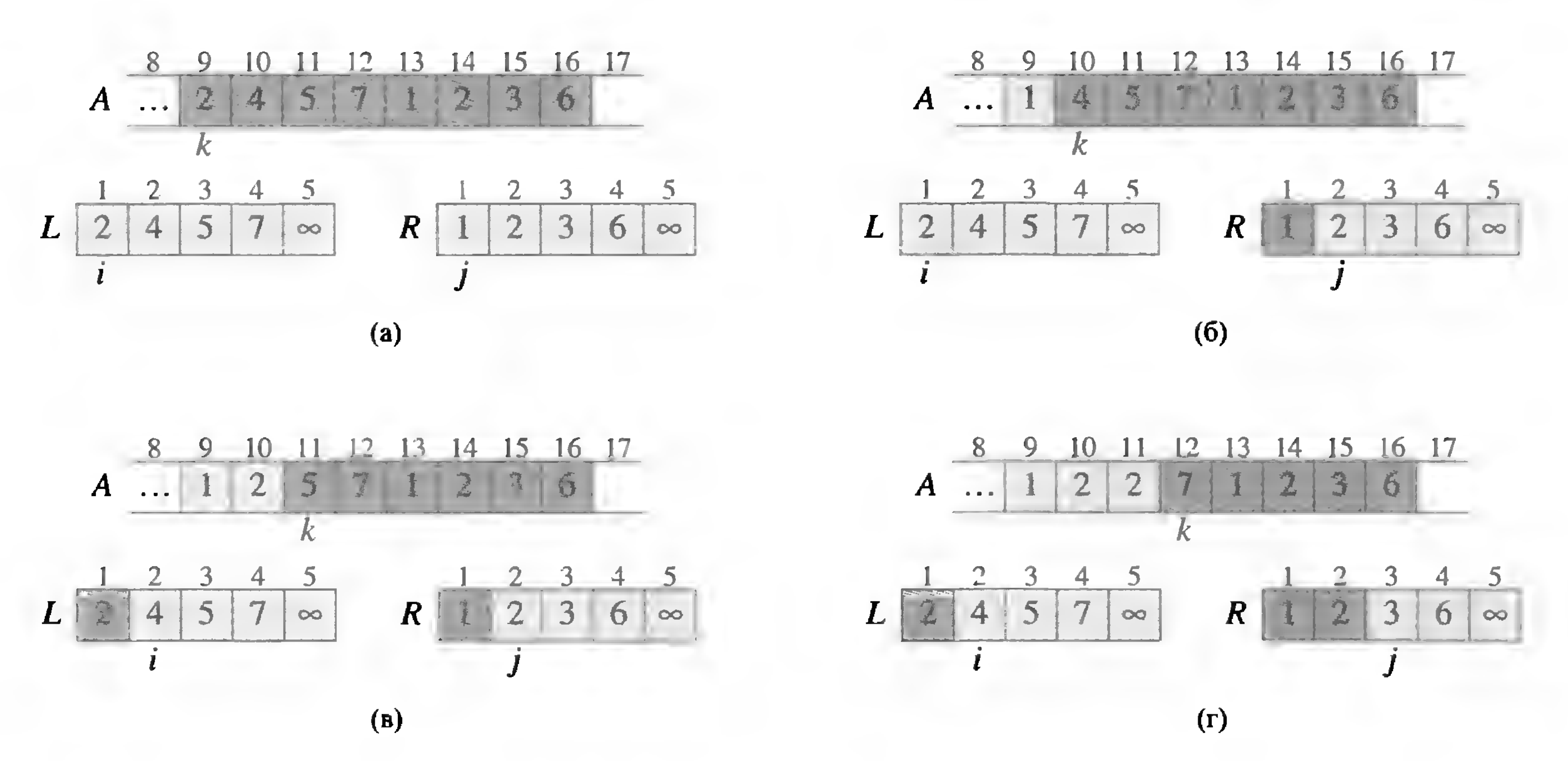


Рис. 2.3. Операции в строках 10–17 процедуры MERGE$(A, 9, 12, 16)$, когда подмассив $A[9..16]$ содержит последовательность $\langle 2,4,5,7,1,2,3,6 \rangle$. После копирования и добавления ограничителей массив L содержит $\langle 2, 4, 5, 7, \infty \rangle$, а массив R содержит $\langle 1, 2, 3, 6, \infty \rangle$. Слабо заштрихованные ячейки A содержат окончательные значения, а такие же слабо заштрихованные ячейки в L и R — значения, которые должны быть скопированы назад в A. Вместе слабо заштрихованные ячейки всегда включают значения, изначально находившиеся в $A[9..16]$, вместе с двумя ограничителями. Сильно заштрихованные ячейки A содержат значения, в которые будет выполнено перезаписывающее копирование, а такие же сильно заштрихованные ячейки в L и R содержат значения, уже скопированные обратно в A. **(а)–(з)** Массивы A, L и R и их индексы k, i и j перед каждой итерацией цикла в строках 12–17.

в ходе которого вычисляется индекс q, разделяющий массив $A[p..r]$ на два подмассива: $A[p..q]$ с $\lceil n/2 \rceil$ элементами и $A[q+1..r]$ с $\lfloor n/2 \rfloor$ элементами.[9]

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Для сортировки всей последовательности $A = \langle A[1], A[2], \ldots, A[n] \rangle$ вызывается процедура MERGE-SORT$(A, 1, A.length)$, где $A.length = n$. На рис. 2.4 проиллюстрирована работа этой процедуры в восходящем направлении, когда n представляет собой степень двойки. В ходе работы алгоритма происходит попарное объединение одноэлементных последовательностей в отсортированные последо-

[9]Выражение $\lceil x \rceil$ обозначает наименьшее целое число, которое больше или равно x, а выражение $\lfloor x \rfloor$ — наибольшее целое число, которое меньше или равно x. Эти обозначения вводятся в главе 3. Чтобы убедиться в том, что в результате присваивания переменной q значения $\lfloor (p+r)/2 \rfloor$ получаются подмассивы $A[p..q]$ и $A[q+1..r]$ с размерами $\lceil n/2 \rceil$ и $\lfloor n/2 \rfloor$, достаточно проверить четыре возможных случая, в которых каждое из чисел p и r либо четное, либо нечетное.

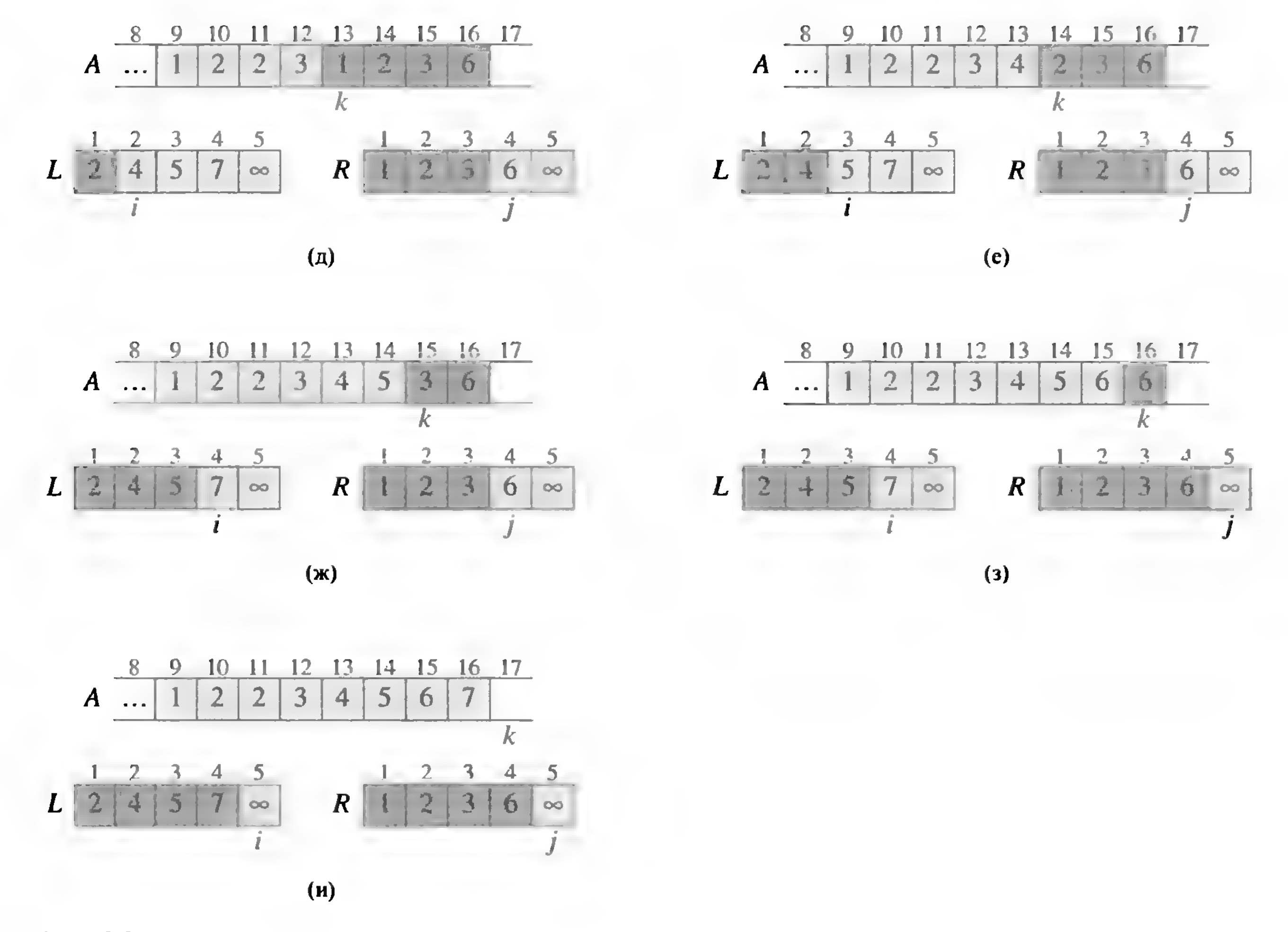


Рис. 2.3 (продолжение). **(и)** Массивы и индексы при завершении процедуры. В этот момент подмассив $A[9..16]$ отсортирован, а два ограничителя в L и R являются единственными элементами в этих массивах, которые не были скопированы в A.

вательности длиной 2, затем — попарное объединение двухэлементных последовательностей в отсортированные последовательности длиной 4 и так до тех пор, пока не будут получены две последовательности, состоящие из $n/2$ элементов, которые объединяются в конечную отсортированную последовательность длиной n.

2.3.2. Анализ алгоритмов, основанных на принципе "разделяй и властвуй"

Когда алгоритм рекурсивно вызывает сам себя, зачастую время его работы можно описать с помощью ***рекуррентного уравнения*** (или ***рекуррентности***), которое выражает полное время, требующееся для решения всей задачи размером n, через время решения задач для меньших входных данных. Затем можно прибегнуть к соответствующему математическому аппарату для решения рекуррентности и получить границы производительности алгоритма.

Получение рекуррентного соотношения для времени работы алгоритма, основанного на принципе "разделяй и властвуй", базируется на трех этапах, соответствующих парадигме этого принципа. Как и ранее, обозначим через $T(n)$ время решения задачи, размер которой равен n. Если размер задачи достаточно мал, скажем, $n \leq c$ для некоторой заранее известной константы c, то задача решается непосредственно в течение определенного константного времени, которое мы обозначим через $\Theta(1)$. Предположим, что наша задача делится на a подзадач,

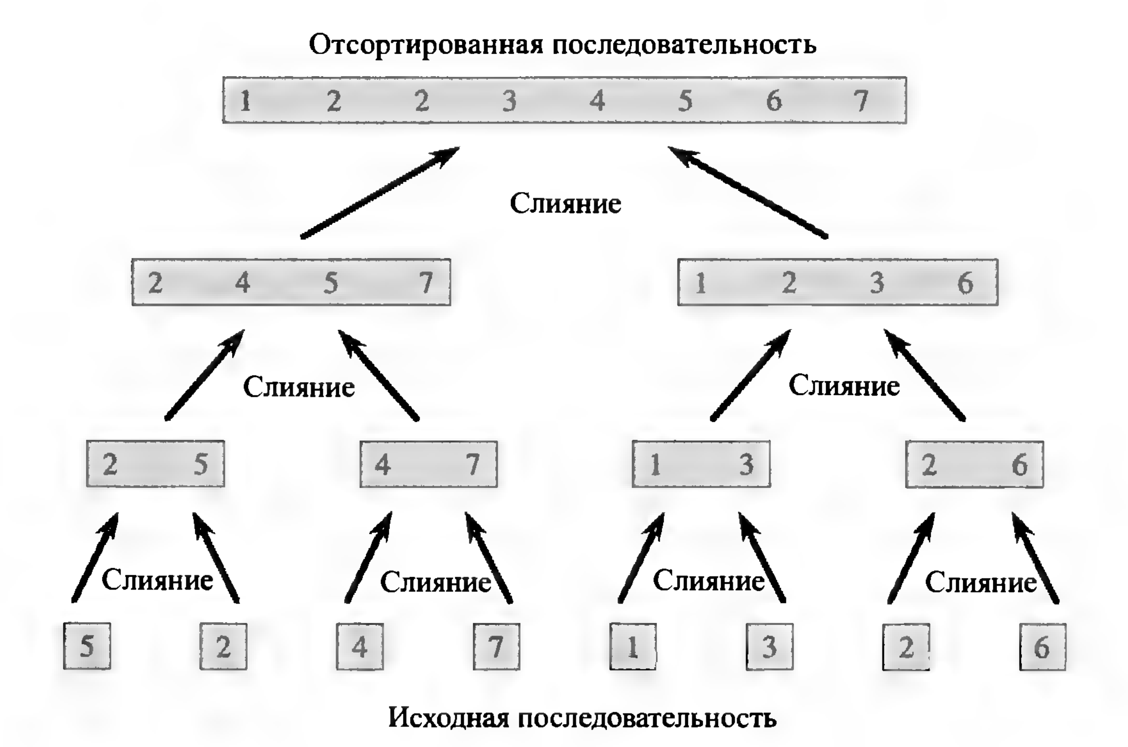


Рис. 2.4. Процесс сортировки слиянием массива $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Длины подлежащих слиянию отсортированных последовательностей возрастают в ходе работы алгоритма.

объем каждой из которых равен $1/b$ от объема исходной задачи. (В алгоритме сортировки методом слияния числа a и b были равны 2, однако нам предстоит ознакомиться со многими алгоритмами разбиения, в которых $a \neq b$.) Для решения подзадачи размером n/b требуется время $T(n/b)$, а для решения a таких подзадач требуется время $aT(n/b)$. Если разбиение задачи на вспомогательные подзадачи происходит за время $D(n)$, а объединение решений подзадач в решение исходной задачи — в течение времени $C(n)$, то мы получим следующее рекуррентное соотношение:

$$T(n) = \begin{cases} \Theta(1), & \text{если } n \leq c \ , \\ aT(n/b) + D(n) + C(n) & \text{в противном случае} \ . \end{cases}$$

В главе 4 рассмотрим методы решения рекуррентных уравнений такого вида.

Анализ алгоритма сортировки слиянием

Хотя псевдокод MERGE-SORT корректно работает для нечетного количества сортируемых элементов, анализ рекуррентного уравнения упрощается, если количество элементов в исходной задаче представляет собой степень двойки. В этом случае на каждом шаге деления будут получены две подпоследовательности, размер которых точно равен $n/2$. В главе 4 будет показано, что это предположение не влияет на порядок роста, полученный в результате решения рекуррентного уравнения.

Чтобы получить рекуррентное уравнение для верхней оценки времени работы $T(n)$ алгоритма, выполняющего сортировку n чисел методом слияния, будем рассуждать следующим образом. Сортировка одного элемента методом слияния

выполняется за константное время. Если $n > 1$, время работы распределяется следующим образом.

Разделение. В ходе разбиения определяется, где находится средина подмассива. Эта операция выполняется за константное время, поэтому $D(n) = \Theta(1)$.

Властвование. Рекурсивно решаются две подзадачи, размер каждой из которых составляет $n/2$. Время решения этих подзадач равно $2T(n/2)$.

Комбинирование. Как уже упоминалось, процедура MERGE при работе с n-элементным подмассивом выполняется за время $\Theta(n)$, так что $C(n) = \Theta(n)$.

Складывая функции $D(n)$ и $C(n)$ для анализа алгоритма сортировки слиянием, мы складываем функции, которые представляют собой $\Theta(n)$ и $\Theta(1)$. Эта сумма является линейной функцией от n, т.е. $\Theta(n)$. Добавление ее к члену $2T(n/2)$ из шага "Властвование" дает следующее рекуррентное соотношение для времени работы сортировки слиянием в наихудшем случае:

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1 \ , \\ 2T(n/2) + \Theta(n), & \text{если } n > 1 \ . \end{cases} \tag{2.1}$$

В главе 4 вы познакомитесь с теоремой, которую можно использовать для того, чтобы показать, что $T(n)$ представляет собой $\Theta(n \lg n)$, где $\lg n$ означает $\log_2 n$. Поскольку логарифмическая функция растет медленнее любой линейной функции, для достаточно больших входных данных сортировка слиянием со временем работы $\Theta(n \lg n)$ превзойдет в наихудшем случае сортировку вставкой, время работы которой равно $\Theta(n^2)$.

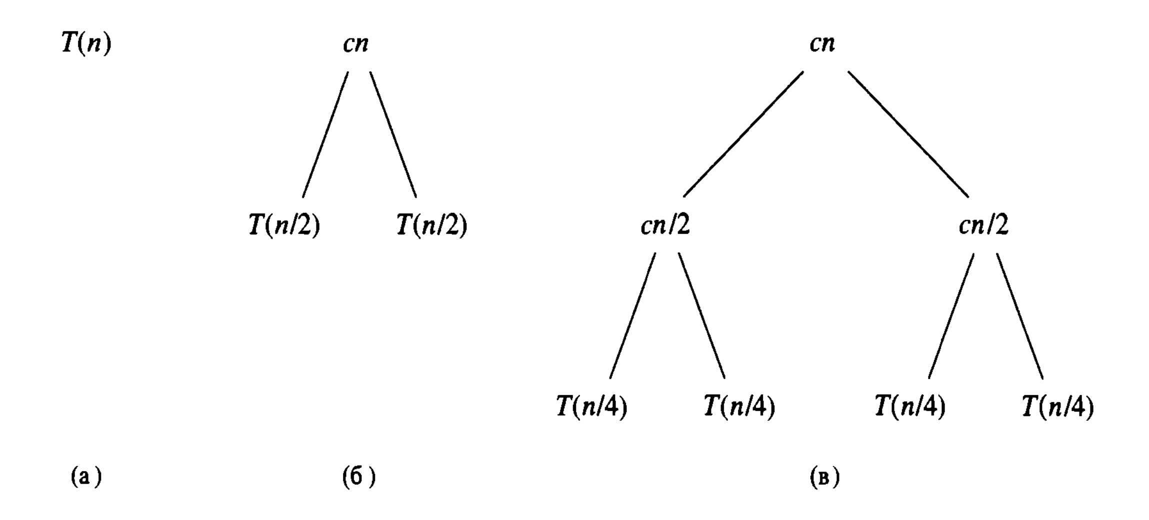
Конечно, можно и без упомянутой теоремы интуитивно понять, почему решением рекуррентного соотношения (2.1) является выражение $T(n) = \Theta(n \lg n)$. Давайте перепишем рекуррентное соотношение (2.1) в виде

$$T(n) = \begin{cases} c, & \text{если } n = 1 \ , \\ 2T(n/2) + cn, & \text{если } n > 1 \ , \end{cases} \tag{2.2}$$

где константа c обозначает время, которое требуется для решения задачи, размер который равен 1, а также удельное (приходящееся на один элемент) время, требуемое для разделения и комбинирования.[10]

На рис. 2.5 показано решение рекуррентного соотношения (2.2). Для удобства мы считаем, что n представляет собой точную степень 2. В части (a) рисунка

[10]Маловероятно, чтобы одна и та же константа представляла и время, необходимое для решения задачи, размер который равен 1, и приходящееся на один элемент время, в течение которого выполняются этапы разбиения и объединения. Чтобы обойти эту проблему, достаточно предположить, что c — максимальный из перечисленных промежутков времени. В таком случае мы получим верхнюю границу времени работы алгоритма. Если же в качестве c выбрать наименьший из всех перечисленных промежутков времени, то в результате решения рекуррентного соотношения можно получить нижнюю границу времени работы алгоритма. Принимая во внимание, что обе границы имеют порядок $n \lg n$, делаем вывод, что время работы алгоритма ведет себя, как $\Theta(n \lg n)$.



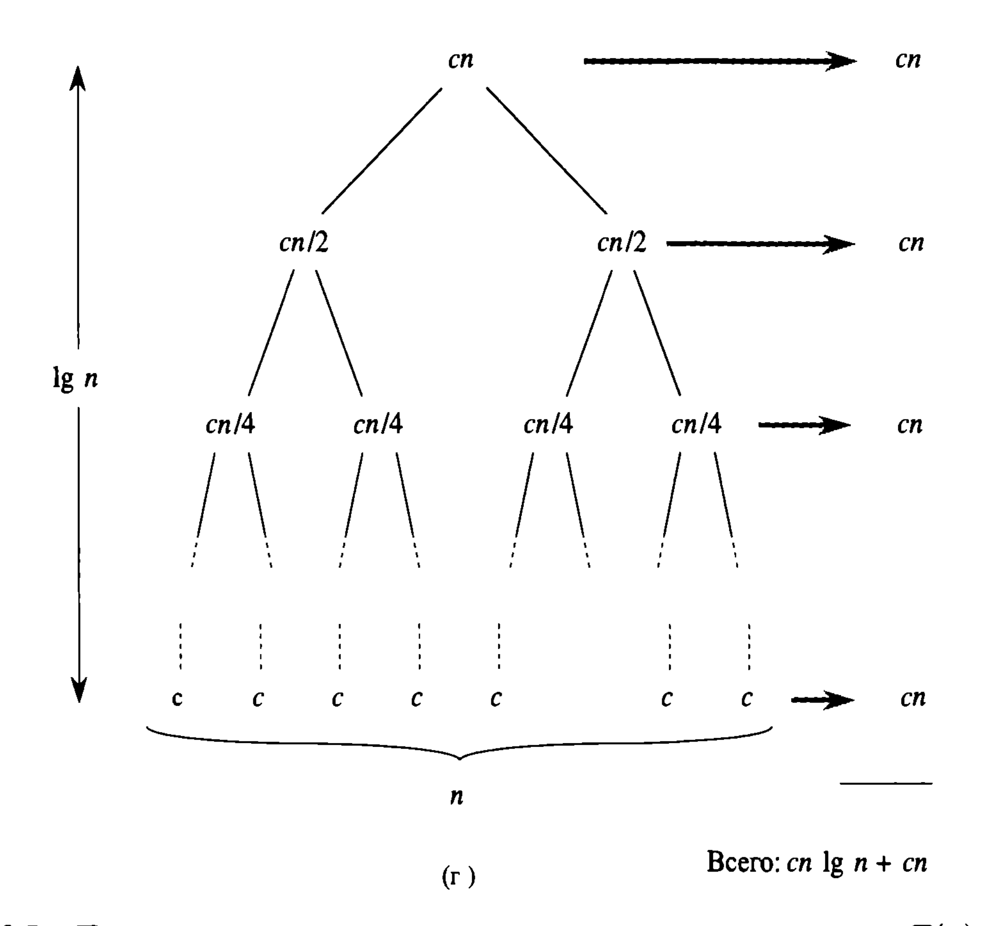


Рис. 2.5. Построение дерева рекурсии для рекуррентного соотношения $T(n) = 2T(n/2) + cn$. В части **(а)** показано время $T(n)$, которое постепенно раскрывается в частях **(б)–(г)**, образуя дерево рекурсии. Полностью раскрытое дерево в части **(г)** имеет $\lg n + 1$ уровней (т.е. его высота, как и указано, равна $\lg n$), а вклад каждого уровня в стоимость равен cn. Таким образом, общая стоимость равна $cn \lg n + cn$, что представляет собой $\Theta(n \lg n)$.

показано время $T(n)$, которое в части (б) представлено в виде эквивалентного дерева, которое представляет рекуррентное уравнение. Корнем дерева является член cn (стоимость на верхнем уровня рекурсии), и от него идут два поддерева, представляющие меньшие рекуррентности $T(n/2)$. В части (в) показан очередной шаг рекурсии, состоящий в раскрытии $T(n/2)$. Стоимость каждого из двух подузлов на втором уровне рекурсии равна $cn/2$. Мы продолжаем раскрывать каждый узел дерева, разбивая его на составные части, определяемые рекуррентным соотношением, пока размеры задач не достигнут значений 1, со стоимостью c. В части (г) показано получающееся в результате ***дерево рекурсии***.

После того как дерево построено, длительности выполнения всех его узлов суммируются по всем уровням. Полное время выполнения верхнего уровня равно cn, следующий уровень дает вклад, равный $c(n/2) + c(n/2) = cn$, общая стоимость уровня после него составляет $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$, и т.д. В общем случае уровень i (если вести отсчет сверху) имеет 2^i узлов, каждый из которых дает вклад в общее время работы алгоритма, равный $c(n/2^i)$, так что общая стоимость i-го уровня равна $2^i\, c(n/2^i) = cn$. На нижнем уровне имеется n узлов, каждый из которых дает вклад c, что в сумме также дает время, равное cn.

Общее количество уровней дерева рекурсии на рис. 2.5 равно $\lg n + 1$, где n — количество листьев, соответствующее размеру входных данных. Это легко понять из неформальных индуктивных рассуждений. В простейшем случае, когда $n = 1$, имеется всего один уровень. Поскольку $\lg 1 = 0$, выражение $\lg n + 1$ дает правильное количество уровней. Теперь в качестве гипотезы индукции примем, что количество уровней рекурсивного дерева с 2^i узлами равно $\lg 2^i + 1 = i + 1$ (так как для любого i выполняется соотношение $\lg 2^i = i$). Поскольку мы предположили, что количество входных элементов равно степени двойки, теперь нужно рассмотреть случай 2^{i+1} элементов. Дерево с 2^{i+1} узлами имеет на один уровень больше, чем дерево с 2^i узлами, поэтому общее количество уровней равно $(i+1)+1 = \lg 2^{i+1} + 1$.

Для вычисления общей стоимости, представленной рекуррентным соотношением (2.2), нужно просто просуммировать вклады от всех уровней. Всего имеется $\lg n + 1$ уровней, каждый из которых имеет стоимость cn, так что полная стоимость составляет $cn(\lg n + 1) = cn\lg n + cn$. Пренебрегая членом более низкого порядка и константой c, в результате получаем $\Theta(n \lg n)$.

Упражнения

2.3.1

Используя в качестве образца рис. 2.4, проиллюстрируйте работу алгоритма сортировки слиянием для массива $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3.2

Перепишите процедуру MERGE так, чтобы в ней не использовались сигнальные значения. Сигналом к остановке должен служить тот факт, что все элементы массива L или массива R скопированы обратно в массив A, после чего в этот массив копируются элементы, оставшиеся в непустом массиве.

2.3.3

Воспользуйтесь методом математической индукции для доказательства того, что, когда n является точной степенью 2, решением рекуррентного соотношения

$$T(n) = \begin{cases} 2, & \text{если } n = 2\ , \\ 2T(n/2) + n, & \text{если } n = 2^k,\ k > 1, \end{cases}$$

является $T(n) = n \lg n$.

2.3.4

Сортировку вставкой можно представить в виде рекурсивной последовательности. Чтобы отсортировать массив $A[1\,..\,n]$, сначала нужно рекурсивно отсортировать массив $A[1\,..\,n-1]$, после чего в этот отсортированный массив помещается элемент $A[n]$. Запишите рекуррентное уравнение для времени работы этой рекурсивной версии сортировки вставкой.

2.3.5

Возвращаясь к задаче поиска (см. упр. 2.1.3), нетрудно заметить, что если последовательность A отсортирована, то можно сравнить значение среднего элемента этой последовательности с искомым значением v и сразу исключить половину последовательности из дальнейшего рассмотрения. ***Бинарный поиск*** (binary search) — это алгоритм, в котором такая процедура повторяется неоднократно, что всякий раз приводит к уменьшению оставшейся части последовательности в два раза. Запишите псевдокод алгоритма бинарного поиска (либо итеративный, либо рекурсивный). Докажите, что время работы этого алгоритма в наихудшем случае составляет $\Theta(\lg n)$.

2.3.6

Заметим, что в цикле **while** в строках 5–7 процедуры INSERTION-SORT в разделе 2.1 для сканирования (в обратном порядке) отсортированного подмассива $A[1\,..\,j-1]$ используется линейный поиск. Можно ли использовать бинарный поиск (см. упр. 2.3.5) вместо линейного, чтобы время работы этого алгоритма в наихудшем случае улучшилось и стало равным $\Theta(n \lg n)$?

2.3.7 ★

Разработайте алгоритм со временем работы $\Theta(n \lg n)$, который для заданного множества S из n целых чисел и другого целого числа x определяет, имеются ли в множестве S два элемента, сумма которых равна x.

Задачи

2.1. Сортировка вставкой малых массивов в процессе сортировки слиянием

Несмотря на то что с увеличением количества сортируемых элементов время сортировки методом слияний в наихудшем случае растет как $\Theta(n \lg n)$, а время

сортировки вставкой — как $\Theta(n^2)$, благодаря постоянным множителям на практике для малых размеров задач на большинстве машин сортировка вставкой выполняется быстрее. Таким образом, есть смысл использовать сортировку вставок в процессе сортировки методом слияний, когда подзадачи становятся достаточно маленькими. Рассмотрите модификацию алгоритма сортировки слиянием, в котором n/k подмассивов длиной k сортируются вставкой, после чего они объединяются с помощью обычного механизма слияния. Величина k должна быть найдена в процессе решения задачи.

а. Покажите, что сортировка вставкой позволяет отсортировать n/k подпоследовательностей длиной k каждая за время $\Theta(nk)$ в худшем случае.

б. Покажите, как выполнить слияние этих подпоследовательностей за время $\Theta(n \lg(n/k))$ в наихудшем случае.

в. Если такой модифицированный алгоритм выполняется за время $\Theta(nk + n \lg(n/k))$ в наихудшем случае, то чему равно наибольшее значение k как функции от n, для которого модифицированный алгоритм в Θ-обозначениях имеет то же время работы, что и стандартная сортировка слиянием?

г. Как следует выбирать k на практике?

2.2. Корректность пузырьковой сортировки

Пузырьковая сортировка представляет собой популярный, но не эффективный алгоритм сортировки. В его основе лежит многократная перестановка соседних элементов, нарушающих порядок сортировки.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              поменять A[j] и A[j − 1] местами
```

а. Пусть A' обозначает выход процедуры BUBBLESORT(A). Для доказательства корректности процедуры BUBBLESORT необходимо доказать, что она завершается и что

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] , \tag{2.3}$$

где $n = A.length$. Что еще необходимо доказать для того, чтобы показать, что процедура BUBBLESORT действительно выполняет сортировку?

В следующих двух частях доказываются неравенства (2.3).

б. Точно сформулируйте инвариант цикла **for** в строках 2–4 и докажите, что он выполняется. Доказательство должно иметь ту же структуру доказательства инварианта цикла, которая ранее использовалась в аналогичных доказательствах в данной главе.

в. С помощью условия завершения инварианта цикла, доказанного в части (б), сформулируйте инвариант цикла **for** в строках 1–4, который позволил бы доказать неравенства (2.3). Доказательство должно иметь ту же структуру доказательства инварианта цикла, которая использовалась ранее в аналогичных доказательствах в данной главе.

г. Определите время пузырьковой сортировки в наихудшем случае и сравните его со временем сортировки вставкой.

2.3. Корректность правила Горнера

Следующий фрагмент кода реализует правило Горнера для вычисления полинома

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots))$$

для заданных коэффициентов $a_0, a_1, \ldots, a_n$ и значения x.

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

а. Чему равно время работы этого фрагмента кода правила Горнера в Θ-обозначениях?

б. Напишите псевдокод, реализующий алгоритм обычного вычисления полинома, когда каждое слагаемое полинома вычисляется отдельно. Определите асимптотическое время работы этого алгоритма и сравните его со временем работы алгоритма, основанного на правиле Горнера.

в. Рассмотрим следующий инвариант цикла.

В начале каждой итерации цикла **for** в строках 2 и 3

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Рассматривайте сумму без членов как равную нулю. Следуя структуре доказательства инварианта цикла, которая использовалась ранее в данной главе, воспользуйтесь указанным инвариантом цикла, чтобы показать, что по завершении работы $y = \sum_{k=0}^{n} a_k x^k$.

г. Сделайте заключение, что в приведенном фрагменте кода правильно вычисляется значение полинома, который задается коэффициентами $a_0, a_1, \ldots, a_n$.

2.4. Инверсии

Пусть $A[1..n]$ представляет собой массив из n различных чисел. Если $i < j$ и $A[i] > A[j]$, то пара (i, j) называется ***инверсией*** A.

а. Перечислите пять инверсий массива $\langle 2, 3, 8, 6, 1 \rangle$.

б. Какой массив из элементов множества $\{1, 2, \ldots, n\}$ содержит максимальное количество инверсий? Сколько инверсий в этом массиве?

в. Какая существует взаимосвязь между временем сортировки методом вставок и количеством инверсий во входном массиве? Обоснуйте свой ответ.

г. Разработайте алгоритм, определяющий количество инверсий, содержащихся в произвольной перестановке n элементов, время работы которого в наихудшем случае равно $\Theta(n \lg n)$. (*Указание:* модифицируйте алгоритм сортировки слиянием.)

Заключительные замечания

В 1968 году Кнут (Knuth) опубликовал первый из трех томов, объединенных названием *The Art of Computer Programming* (Искусство программирования) [208–210][11]. Этот первый том стал введением в современные компьютерные алгоритмы с акцентом на анализе времени их работы, а весь трехтомник до сих пор остается интереснейшим и ценнейшим пособием по многим темам, представленным в данной книге. Согласно Кнуту, слово "алгоритм" происходит от имени персидского математика IX века аль-Хорезми (al-Khowârizmî).

Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] являются сторонниками асимптотического анализа алгоритмов (с использованием обозначений, вводимых в главе 3, включая Θ-обозначения), являющегося средством сравнения их относительной производительности. Они также популяризируют применение рекуррентных соотношений для описания времени работы рекурсивных алгоритмов.

Кнут [210] с энциклопедической полнотой рассмотрел многие алгоритмы сортировки. Его сравнение алгоритмов сортировки включает в себя подробный анализ с точным подсчетом шагов, подобный проведенному в этой книге для сортировки методом вставок. В ходе обсуждения Кнутом алгоритма сортировки вставкой приводится несколько вариаций этого алгоритма. Важнейшей из них является сортировка Д.Л. Шелла (D.L. Shell), который использовал сортировку методом

[11]Имеется русский перевод этих книг: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000; Д. Кнут. *Искусство программирования, т. 2. Получисленные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000; Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000. Кроме того, уже после написания данной книги вышел очередной том "*Искусства программирования*": Д. Кнут. *Искусство программирования, т. 4, А. Комбинаторные алгоритмы, часть 1.* — М.: И.Д. "Вильямс", 2013.

вставок для упорядочения периодических подпоследовательностей, чтобы получить более производительный алгоритм сортировки.

В книге Кнута также описана сортировка слиянием. В ней же упоминается, что в 1938 году была создана механическая машина, способная за один проход объединять две стопки перфокарт. По всей видимости, первым программу для сортировки методом слияния (предназначенную для компьютера EDVAC) в 1945 году разработал Джон фон Нейман (J. von Neumann), который был одним из создателей теории вычислительных машин.

Ранние этапы развития в области доказательства корректности программ описаны Гризом (Gries) [152], который считает, что первая статья по этой теме была написана П. Науром (P. Naur). Авторство понятия инварианта цикла Гриз приписывает Р. Флойду (R. W. Floyd). В учебнике Митчелла (Mitchell) [254] описывается прогресс, достигнутый в области доказательства корректности программ в настоящее время.

Глава 3. Рост функций

Определенный в главе 2 порядок роста, характеризующий время работы алгоритма, является наглядным показателем эффективности алгоритма, а также позволяет сравнивать производительность различных алгоритмов. Если количество сортируемых элементов n становится достаточно большим, производительность алгоритма сортировки по методу слияний, время работы которого в наихудшем случае возрастает как $\Theta(n \lg n)$, становится выше производительности алгоритма сортировки вставкой, время работы которого в наихудшем случае возрастает как $\Theta(n^2)$. Несмотря на то что в некоторых случаях можно определить точное время работы алгоритма, как это было сделано для алгоритма сортировки методом вставок в главе 2, обычно не стоит тратить лишние усилия для получения оценки с высокой точностью. Для достаточно больших входных данных постоянные множители и слагаемые низшего порядка, фигурирующие в выражении для точного времени работы алгоритма, подавляются эффектами, вызванными увеличением размера входных данных.

Рассматривая входные данные достаточно больших размеров для оценки только такой величины, как порядок роста времени работы алгоритма, мы тем самым изучаем ***асимптотическую*** эффективность алгоритмов. Это означает, что нас интересует только то, как время работы алгоритма растет с увеличением размера входных данных *в пределе*, когда этот размер увеличивается до бесконечности. Обычно алгоритм, более эффективный в асимптотическом смысле, будет более производительным для всех входных данных, за исключением очень маленьких.

В этой главе приводится несколько стандартных методов, позволяющих упростить асимптотический анализ алгоритмов. В начале следующего раздела вводится несколько видов "асимптотических обозначений", одним из которых являются уже известные нам Θ-обозначения. Далее представлены некоторые соглашения по поводу обозначений, принятых в данной книге. Наконец, в завершающей части главы, рассматривается поведение функций, часто встречающихся в ходе анализа алгоритмов.

3.1. Асимптотические обозначения

Обозначения, используемые нами для описания асимптотического поведения времени работы алгоритма, используют функции, областью определения которых является множество неотрицательных целых чисел $\mathbb{N} = \{0, 1, 2, \ldots\}$. Подобные обозначения удобны для описания времени работы $T(n)$ в наихудшем случае как функции, определенной только для целых чисел, представляющих собой размер входных данных. Однако иногда удобно изменить толкование асимптотических обозначений тем или иным не совсем корректным образом. Например, можно распространить эти обозначения на действительные числа или, наоборот, ограничить их областью, являющейся подмножеством натуральных чисел. При этом важно понимать точный смысл обозначений, чтобы изменение толкования не привело к *некорректному* их использованию. В данном разделе вводятся основные асимптотические обозначения, а также описывается, как чаще всего неверно изменяется их толкование.

Асимптотические обозначения, функции и время работы

Асимптотические обозначения будут использоваться, в первую очередь, для описания времени работы алгоритмов, как, например, мы записывали время работы сортировки вставкой в наихудшем случае как $\Theta(n^2)$. Однако фактически асимптотические обозначения применяются к функциям. Вспомним, что мы охарактеризовали время работы сортировки вставкой в наихудшем случае как $an^2 + bn + c$ для некоторых констант a, b и c. Записывая время работы сортировки вставкой как $\Theta(n^2)$, мы абстрагируемся от некоторых деталей этой функции. Поскольку асимптотические обозначения применимы к функциям, то то, что мы записываем просто как $\Theta(n^2)$, представляет собой функцию $an^2 + bn + c$, которая в данном случае, помимо прочего, описывает время работы сортировки вставкой в наихудшем случае.

В этой книге функции, к которым мы применяем асимптотические обозначения, обычно характеризуют время работы алгоритма. Но эти обозначения могут применяться и к функциям, которые характеризуют некоторые другие аспекты алгоритмов (например, количество используемой памяти), или даже к функциям, не имеющим никакого отношения к алгоритмам.

Даже когда мы используем асимптотические обозначения для времени работы алгоритма, нам надо понимать, *какое* время работы мы имеем в виду. В основном нас интересует время работы в наихудшем случае, однако зачастую мы хотим охарактеризовать время работы независимо от входных данных. Другими словами, часто желательно получить всеобъемлющее утверждение, охватывающее все варианты входных данных, а не только наихудший случай. Мы познакомимся с асимптотическими обозначениями, которые хорошо подходят для характеристики времени выполнения безотносительно ко входным данным.

Θ-обозначения

В главе 2 было показано, что время работы сортировки вставкой в наихудшем случае составляет $T(n) = \Theta(n^2)$. Давайте разберемся в смысле данного обозначения. Для заданной функции $g(n)$ запись $\Theta(g(n))$ обозначает *множество функций*

$$\Theta(g(n)) = \{f(n) : \text{существуют положительные константы } c_1, c_2 \text{ и } n_0, \text{ такие, что } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ для всех } n \ge n_0\} .^1$$

Функция $f(n)$ принадлежит множеству $\Theta(g(n))$, если существуют положительные константы c_1 и c_2, такие, что при достаточно больших n эта функция может быть заключена в рамки между $c_1 g(n)$ и $c_2 g(n)$. Поскольку $\Theta(g(n))$ представляет собой множество, можно записать "$f(n) \in \Theta(g(n))$", чтобы указать тот факт, что $f(n)$ является членом $\Theta(g(n))$. Вместо этого мы обычно будем использовать эквивалентную запись "$f(n) = \Theta(g(n))$". Такое толкование знака равенства для обозначения принадлежности множеству поначалу может сбить с толку, однако далее мы убедимся, что у этой записи есть свои преимущества.

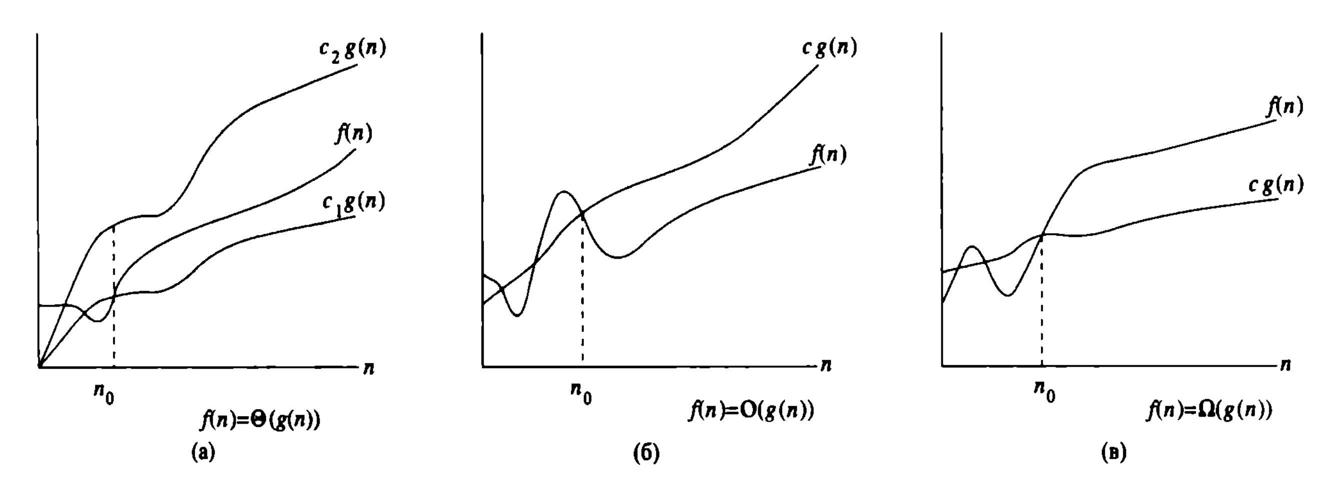


Рис. 3.1. Графические примеры Θ-, O- и Ω-обозначений. В каждой части значение n_0 является минимально возможным; подходит также любое большее значение. **(а)** Θ-обозначение ограничивает функцию константными множителями. Мы записываем $f(n) = \Theta(g(n))$, если существуют положительные константы n_0, c_1 и c_2, такие, что в точке n_0 и справа от нее значение $f(n)$ всегда лежит между $c_1 g(n)$ и $c_2 g(n)$ включительно. **(б)** O-обозначение дает верхнюю оценку функции с точностью до постоянного множителя. Мы записываем $f(n) = O(g(n))$, если существуют положительные константы n_0 и c, такие, что в точке n_0 и справа от нее значение $f(n)$ всегда лежит не выше $cg(n)$. **(в)** Ω-обозначение дает нижнюю оценку функции с точностью до постоянного множителя. Мы записываем $f(n) = \Omega(g(n))$, если существуют положительные константы n_0 и c, такие, что в точке n_0 и справа от нее значение $f(n)$ всегда лежит не ниже $cg(n)$.

На рис. 3.1, (а) показано интуитивное изображение функций $f(n)$ и $g(n)$, таких, что $f(n) = \Theta(g(n))$. Для всех значений n, лежащих справа от n_0, функция $f(n)$ больше или равна функции $c_1 g(n)$, но не превосходит функцию $c_2 g(n)$. Другими словами, для всех $n \ge n_0$ функция $f(n)$ равна функции $g(n)$ с точностью до

---

[1] В теории множеств двоеточие следует читать как "такие, что" или "для которых выполняется условие".

постоянного множителя. Говорят, что функция $g(n)$ является ***асимптотически точной оценкой*** функции $f(n)$.

Согласно определению множества $\Theta(g(n))$ необходимо, чтобы каждый элемент $f(n) \in \Theta(g(n))$ этого множества был ***асимптотически неотрицателен***. Это означает, что при достаточно больших n функция $f(n)$ является неотрицательной. (***Асимптотически положительной*** называется такая функция, которая является положительной при любых достаточно больших n.) Следовательно, функция $g(n)$ должна быть асимптотически неотрицательной, потому что в противном случае множество $\Theta(g(n))$ окажется пустым. Поэтому будем считать, что все функции, используемые в Θ-обозначениях, асимптотически неотрицательные. Это предположение справедливо и для других асимптотических обозначений, определенных в данной главе.

В главе 2 Θ-обозначения вводятся неформально. При этом отбрасываются слагаемые низшего порядка и игнорируется коэффициент при старшем слагаемом. Подтвердим интуитивные представления, рассмотрев небольшой пример, в котором с помощью формального определения показано, что $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Для этого необходимо определить, чему равны положительные константы c_1, c_2 и n_0, такие, что

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

для всех $n \geq n_0$. Деление на n^2 дает

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

Можно сделать правое неравенство выполняющимся для любого значения $n \geq 1$, выбирая любую константу $c_2 \geq 1/2$. Аналогично можно сделать выполняющимся для любого значения $n \geq 7$ левое неравенство, если выбрать произвольную константу $c_1 \leq 1/14$. Таким образом, выбирая $c_1 = 1/14$, $c_2 = 1/2$ и $n_0 = 7$, можно убедиться, что $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Конечно, имеются и другие варианты выбора констант, но главное заключается в том, что такой выбор *существует*. Заметим, что эти константы зависят от функции $\frac{1}{2}n^2 - 3n$; другой функции, принадлежащей $\Theta(n^2)$, скорее всего, потребуются другие константы.

Можно воспользоваться формальным определением для того, чтобы убедиться, что $6n^3 \neq \Theta(n^2)$. Пойдем от противного, предположив, что существуют константы c_2 и n_0, такие, что $6n^3 \leq c_2 n^2$ для всех $n \geq n_0$. Но тогда деление на n^2 дает $n \leq c_2/6$, а это неравенство не может выполняться при произвольно больших n, поскольку c_2 является константой.

Интуитивно понятно, что при асимптотически точной оценке асимптотически положительных функций, слагаемыми низших порядков в них можно пренебречь, поскольку при больших n они становятся несущественными. При больших n даже небольшой доли слагаемого самого высокого порядка достаточно для того, чтобы превзойти слагаемые низших порядков. Таким образом, для выполнения неравенств, фигурирующих в определении Θ-обозначений, достаточно в качестве c_1 выбрать значение, которое несколько меньше коэффициента при самом старшем слагаемом, а в качестве c_2 — значение, которое несколько больше этого ко-

эффициента. Поэтому коэффициент при старшем слагаемом можно не учитывать, так как он лишь изменяет указанные константы.

В качестве примера рассмотрим произвольную квадратичную функцию $f(n) = an^2 + bn + c$, где a, b и c — константы, причем $a > 0$. Отбросив слагаемые низших порядков и проигнорировав константу, получим $f(n) = \Theta(n^2)$. Чтобы показать то же самое формально, выберем константы $c_1 = a/4$, $c_2 = 7a/4$ и $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. Читатель может сам убедиться в том, что неравенство $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ выполняется для всех $n \geq n_0$. В общем случае для любого полинома $p(n) = \sum_{i=0}^{d} a_i n^i$, где a_i представляют собой константы и $a_d > 0$, мы имеем $p(n) = \Theta(n^d)$ (см. задачу 3.1).

Поскольку любая константа — это полином нулевой степени, постоянную функцию можно выразить как $\Theta(n^0)$ или $\Theta(1)$. Однако последнее обозначение не совсем точное, поскольку непонятно, по отношению к какой переменной исследуется асимптотика[2]. Мы часто будем употреблять запись $\Theta(1)$ для обозначения либо константы, либо постоянной функции по отношению к некоторой переменной.

O-обозначения

В Θ-обозначениях функция асимптотически ограничивается сверху и снизу. Если же достаточно определить только ***асимптотическую верхнюю границу***, используются O-обозначения. Для данной функции $g(n)$ обозначение $O(g(n))$ (произносится как "о большое от g от n" или просто "о от g от n") означает множество функций

$$O(g(n)) = \{f(n) : \text{существуют положительные константы } c \text{ и } n_0, \text{ такие, что } 0 \leq f(n) \leq cg(n) \text{ для всех } n \geq n_0\} .$$

O-обозначения применяются, когда нужно указать верхнюю границу функции с точностью до постоянного множителя. Интуитивное представление об O-обозначениях позволяет получить рис. 3.1, (б). Для всех значений n, лежащих справа от n_0, значение функции $f(n)$ не превышает значения функции $cg(n)$.

Чтобы указать, что функция $f(n)$ принадлежит множеству $O(g(n))$, используется запись $f(n) = O(g(n))$. Обратите внимание, что из $f(n) = \Theta(g(n))$ следует $f(n) = O(g(n))$, поскольку Θ-обозначения более сильные, чем O-обозначения. В обозначениях теории множеств $\Theta(g(n)) \subseteq O(g(n))$. Таким образом, доказательство того, что функция $an^2 + bn + c$, где $a > 0$, принадлежит множеству $\Theta(n^2)$, одновременно доказывает, что любая такая квадратичная функция является элементом множества $O(n^2)$. Может показаться удивительным то, что любая *линейная* функция $an + b$ при $a > 0$ также принадлежит множеству $O(n^2)$. В этом легко убедиться, выбрав $c = a + |b|$ и $n_0 = \max(1, -b/a)$.

---

[2]На самом деле проблема заключается в том, что в наших обычных обозначениях функций не делается различия между функциями и обычными величинами. В λ-исчислении четко указываются параметры функций: функцию от n^2 можно обозначить как $\lambda n.n^2$ или даже как $\lambda r.r^2$. Однако если принять более строгие обозначения, то алгебраические преобразования могут усложниться, поэтому мы предпочли нестрогие обозначения.

Некоторым читателям, уже знакомым с O-обозначениями, может показаться странным, например, соотношение $n = O(n^2)$. В литературе O-обозначения иногда неформально используются для описания асимптотической точной оценки, т.е. так, как мы определили Θ-обозначения. Однако в данной книге, когда мы пишем $f(n) = O(g(n))$, подразумевается, что произведение некоторой константы на функцию $g(n)$ является асимптотическим верхним пределом функции $f(n)$. При этом не играет роли, насколько близко функция $f(n)$ находится к этой верхней границе. В литературе, посвященной алгоритмам, стало стандартом различать асимптотически точную оценку и верхнюю асимптотическую границу.

Чтобы записать время работы алгоритма в O-обозначениях, зачастую достаточно просто изучить его общую структуру. Например, наличие двойного вложенного цикла в структуре алгоритма сортировки вставкой, представленного в главе 2, свидетельствует о том, что верхний предел времени работы в наихудшем случае выражается как $O(n^2)$: стоимость каждой итерации во внутреннем цикле ограничена сверху константой $O(1)$, индексы i и j — числом n, а внутренний цикл выполняется самое большее один раз для каждой из n^2 пар значений i и j.

Поскольку O-обозначения описывают верхнюю границу, когда мы используем их для ограничения времени работы алгоритма в наихудшем случае, мы получаем верхнюю границу этой величины для любых входных данных — то самое всеобъемлющее утверждение, о котором мы говорили ранее. Таким образом, граница $O(n^2)$ для времени работы алгоритма в наихудшем случае применима для времени решения задачи с любыми входными данными, чего нельзя сказать о Θ-обозначениях. Например, оценка $\Theta(n^2)$ для времени сортировки вставкой в наихудшем случае неприменима для *любых* входных данных. Например, в главе 2 мы имели возможность убедиться в том, что если входные элементы уже отсортированы, то время работы алгоритма сортировки вставкой составляет $\Theta(n)$.

Технически неверно говорить, что время, необходимое для сортировки по методу вставок, равно $O(n^2)$, так как для данного n фактическое время работы алгоритма изменяется в зависимости от конкретных входных данных. Когда говорится, что "время работы равно $O(n^2)$", то подразумевается, что существует функция $f(n)$, принадлежащая $O(n^2)$, такая, что при любых входных данных размером n время решения задачи с этими входными данными ограничено сверху значением функции $f(n)$. В конечном счете подразумевается, что в наихудшем случае время работы равно $O(n^2)$.

Ω-обозначения

Подобно тому, как в O-обозначениях дается асимптотическая *верхняя* граница функции, в Ω-обозначениях дается ее ***асимптотическая нижняя граница***. Для данной функции $g(n)$ выражение $\Omega(g(n))$ (произносится как "омега большое от g от n" или просто как "омега от g от n") обозначает множество функций

$$\Omega(g(n)) = \{f(n) : \text{существуют положительные константы } c \text{ и } n_0, \text{ такие, что } 0 \le cg(n) \le f(n) \text{ для всех } n \ge n_0\} .$$

Интуитивное представление об Ω-обозначениях дает рис. 3.1, (в). Для всех n, лежащих справа от n_0, значения функции $f(n)$ больше или равны значениям $cg(n)$.

Пользуясь введенными определениями асимптотических обозначений, легко доказать сформулированную ниже теорему (см. упр. 3.1.5).

Теорема 3.1

Для любых двух функций $f(n)$ и $g(n)$ мы имеем $f(n) = \Theta(g(n))$ тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. ■

В качестве примера применения этой теоремы отметим, что из соотношения $an^2 + bn + c = \Theta(n^2)$ для произвольных констант a, b и c, где $a > 0$, непосредственно следует, что $an^2 + bn + c = O(n^2)$ и $an^2 + bn + c = \Omega(n^2)$. На практике теорема 3.1 применяется не для получения асимптотических верхней и нижней границ, как это сделано выше, а наоборот — для определения асимптотически точной оценки с помощью асимптотических верхней и нижней границ.

Когда мы говорим, что *время работы* (без указания модификатора) алгоритма равно $\Omega(g(n))$, мы имеем в виду, что *независимо от того, какие конкретно входные данные размером n выбраны для каждого значения n*, время работы для этих входных данных будет для достаточно больших n как минимум константой, умноженной на $g(n)$. Или, что то же самое, мы даем нижнюю границу времени работы алгоритма в наилучшем случае. Например, в наилучшем случае время работы сортировки вставкой составляет $\Omega(n)$, откуда следует, что время работы сортировки вставкой представляет собой $\Omega(n)$.

Таким образом, время работы алгоритма сортировки вставкой принадлежит как $\Omega(n)$, так и $O(n^2)$, поскольку оно располагается между линейной и квадратичной функциями от n. Более того, эти границы охватывают асимптотику настолько плотно, насколько это возможно. Например, нижняя оценка для времени работы алгоритма сортировки вставкой не может быть равной $\Omega(n^2)$, потому что существуют входные данные, для которых эта сортировка выполняется за время $\Theta(n)$ (когда входные элементы уже отсортированы). Это не противоречит утверждению о том, что время работы алгоритма сортировки вставкой в наихудшем случае равно $\Omega(n^2)$, поскольку существуют входные данные, для которых этот алгоритм работает в течение времени $\Omega(n^2)$.

Асимптотические обозначения в уравнениях и неравенствах

Мы уже видели, как асимптотические обозначения используются в математических формулах. Например, при введении O-обозначения мы писали "$n = O(n^2)$". Можно также записать "$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$". Как же интерпретируются подобные формулы?

Если в правой части уравнения (или неравенства) находится только асимптотическое обозначение (не являющееся частью большей формулы), как в случае уравнения $n = O(n^2)$, то знак равенства используется для указания принадлежности множеству: $n \in O(n^2)$. Однако если асимптотические обозначения встречаются в формуле в другой ситуации, они рассматриваются как подставляемые вместо некоторой неизвестной функции, имя которой не имеет значения. Напри-

мер, формула $2n^2+3n+1 = 2n^2+\Theta(n)$ означает, что $2n^2+3n+1 = 2n^2+f(n)$, где $f(n)$ — некоторая функция из множества $\Theta(n)$. В данном случае $f(n) = 3n + 1$, и эта функция действительно принадлежит множеству $\Theta(n)$.

Подобное использование асимптотических обозначений позволяет избежать несущественных деталей и неразберихи в уравнениях. Например, в главе 2 время работы сортировки слиянием в наихудшем случае было выражено в виде рекуррентного уравнения

$$T(n) = 2T(n/2) + \Theta(n) .$$

Если нас интересует только асимптотическое поведение $T(n)$, то нет смысла точно выписывать все слагаемые низших порядков; подразумевается, что все они включены в безымянную функцию, обозначенную как $\Theta(n)$.

Предполагается, что таких функций в выражении столько, сколько раз в нем встречаются асимптотические обозначения. Например, в выражении $\sum_{i=1}^{n} O(i)$ имеется только одна функция без имени (аргументом которой является i). Таким образом, это выражение — *не то же самое*, что и $O(1) + O(2) + \cdots + O(n)$, выражение, которое действительно не имеет однозначной интерпретации.

В некоторых случаях асимптотические обозначения появляются в левой части уравнения, как, например, в

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

Подобные уравнения интерпретируются в соответствии со следующим правилом: *при любом выборе безымянных функций, подставляемых вместо асимптотических обозначений в левую часть уравнения, можно выбрать и подставить в правую часть такие безымянные функции, что уравнение будет правильным.* Таким образом, наш пример означает, что для *любой* функции $f(n) \in \Theta(n)$ существует *некоторая* функция $g(n) \in \Theta(n^2)$, такая, что $2n^2 + f(n) = g(n)$ для всех n. Другими словами, правая часть уравнения предоставляет меньший уровень детализации, чем левая.

Такие соотношения могут быть объединены в цепочку, как в

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

При этом в соответствии со сформулированными выше правилами можно интерпретировать каждое уравнение отдельно. Согласно первому уравнению существует *некоторая* функция $f(n) \in \Theta(n)$, такая, что для всех n выполняется соотношение $2n^2 + 3n + 1 = 2n^2 + f(n)$. Второе уравнение гласит, что для любой функции $g(n) \in \Theta(n)$ (такой, как только что упоминавшаяся $f(n)$) существует *некоторая* функция $h(n) \in \Theta(n^2)$, такая, что для всех n выполняется соотношение $2n^2 + g(n) = h(n)$. Заметим, что такая интерпретация подразумевает выполнение соотношения $2n^2 + 3n + 1 = \Theta(n^2)$, которое согласуется с нашими интуитивными представлениями о цепочке уравнений.

o-обозначения

Асимптотическая верхняя граница, предоставляемая O-обозначениями, может описывать асимптотическое поведение функции с разной точностью. Граница $2n^2 = O(n^2)$ дает правильное представление об асимптотическом поведении функции, а граница $2n = O(n^2)$ — нет. Для указания того, что верхняя граница не является асимптотически точной оценкой функции, применяются o-обозначения. Приведем формальное определение множества $o(g(n))$ (произносится как "о малое от g от n"):

$$o(g(n)) = \{f(n) : \text{для любой положительной константы } c > 0 \text{ существует константа } n_0 > 0, \text{ такая, что } 0 \leq f(n) < cg(n) \text{ для всех } n \geq n_0\} .$$

Например, $2n = o(n^2)$, но $2n^2 \neq o(n^2)$.

Определения O-обозначений и o-обозначений похожи между собой. Основное отличие в том, что определение $f(n) = O(g(n))$ ограничивает функцию $f(n)$ неравенством $0 \leq f(n) \leq cg(n)$ лишь для *некоторой* константы $c > 0$, а определение $f(n) = o(g(n))$ ограничивает ее неравенством $0 \leqslant f(n) < cg(n)$ для *всех* констант $c > 0$. Интуитивно понятно, что в o-обозначениях функция $f(n)$ пренебрежимо мала по сравнению с функцией $g(n)$ при n, стремящемся к бесконечности, т.е.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 . \tag{3.1}$$

Некоторые авторы используют этот предел в качестве определения o-обозначений. Добавим, что определение, данное в этой книге, накладывает на безымянную функцию ограничение, согласно которому она должна быть асимптотически неотрицательной.

ω-обозначения

По аналогии ω-обозначения соотносятся с Ω-обозначениями точно так, как o-обозначения с O-обозначениями. С помощью ω-обозначений указывается нижний предел, не являющийся асимптотически точной оценкой. Один из возможных способов определения ω-обозначения следующий:

$$f(n) \in \omega(g(n)) \text{ тогда и только тогда, когда } g(n) \in o(f(n)) .$$

Формально же $\omega(g(n))$ (произносится как "омега малое от g от n") определяется как множество

$$\omega(g(n)) = \{f(n) : \text{для любой положительной константы } c > 0 \text{ существует константа } n_0 > 0, \text{ такая, что } 0 \leq cg(n) < f(n) \text{ для всех } n \geq n_0\} .$$

Например, $n^2/2 = \omega(n)$, но $n^2/2 \neq \omega(n^2)$. Из соотношения $f(n) = \omega(g(n))$ вытекает, что

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

если этот предел существует. Таким образом, функция $f(n)$ становится сколь угодно большой по сравнению с функцией $g(n)$ при n, стремящемся к бесконечности.

Сравнение функций

Асимптотические сравнения обладают многими свойствами отношений обычных действительных чисел, показанными далее. Здесь предполагается, что функции $f(n)$ и $g(n)$ асимптотически положительны.

Транзитивность

Из $f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ следует $f(n) = \Theta(h(n))$
Из $f(n) = O(g(n))$ и $g(n) = O(h(n))$ следует $f(n) = O(h(n))$
Из $f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ следует $f(n) = \Omega(h(n))$
Из $f(n) = o(g(n))$ и $g(n) = o(h(n))$ следует $f(n) = o(h(n))$
Из $f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$ следует $f(n) = \omega(h(n))$

Рефлексивность

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Симметрия

$f(n) = \Theta(g(n))$ тогда и только тогда, когда $g(n) = \Theta(f(n))$

Перестановочная симметрия

$f(n) = O(g(n))$ тогда и только тогда, когда $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ тогда и только тогда, когда $g(n) = \omega(f(n))$

Из-за выполнения указанных свойств для асимптотических обозначений можно провести аналогию между асимптотическим сравнением двух функций f и g

и сравнением двух действительных чисел a и b.

$$\begin{array}{rll} f(n) = O(g(n)) & \text{аналогично} & a \le b \\ f(n) = \Omega(g(n)) & \text{аналогично} & a \ge b \\ f(n) = \Theta(g(n)) & \text{аналогично} & a = b \\ f(n) = o(g(n)) & \text{аналогично} & a < b \\ f(n) = \omega(g(n)) & \text{аналогично} & a > b \end{array}$$

Говорят, что функция $f(n)$ ***асимптотически меньше*** функции $g(n)$, если $f(n) = o(g(n))$, и ***асимптотически больше*** функции $g(n)$, если $f(n) = \omega(g(n))$.

Однако одно из свойств действительных чисел в асимптотических обозначениях не выполняется.

Трихотомия

Для любых двух действительных чисел a и b должно выполняться только одно из соотношений $a < b$, $a = b$ и $a > b$.

Хотя можно сравнивать любые два действительных числа, в отношении асимптотического сравнения функций это утверждение не является справедливым. Для двух функций $f(n)$ и $g(n)$ может не выполняться ни отношение $f(n) = O(g(n))$, ни отношение $f(n) = \Omega(g(n))$. Например, нельзя асимптотически сравнивать функции n и $n^{1+\sin n}$, поскольку показатель степени в функции $n^{1+\sin n}$ колеблется между значениями 0 и 2, принимая все значения в этом интервале.

Упражнения

3.1.1

Пусть $f(n)$ и $g(n)$ — асимптотически неотрицательные функции. Докажите с помощью базового определения Θ-обозначений, что $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1.2

Покажите, что для любых действительных констант a и b, где $b > 0$, выполняется соотношение

$$(n + a)^b = \Theta(n^b) \ . \tag{3.2}$$

3.1.3

Поясните, почему утверждение "время работы алгоритма A равно как минимум $O(n^2)$" лишено смысла.

3.1.4

Справедливы ли соотношения $2^{n+1} = O(2^n)$ и $2^{2n} = O(2^n)$?

3.1.5

Докажите теорему 3.1.

3.1.6

Докажите, что время работы алгоритма равно $\Theta(g(n))$ тогда и только тогда, когда его время работы в наихудшем случае равно $O(g(n))$, а в наилучшем — $\Omega(g(n))$.

3.1.7

Докажите, что множество $o(g(n)) \cap \omega(g(n))$ является пустым.

3.1.8

Можно обобщить наши обозначения на случай двух параметров n и m, которые могут возрастать до бесконечности по отдельности с разными скоростями. Для данной функции $g(n,m)$ обозначим как $O(g(n,m))$ множество функций

$$O(g(n,m)) = \{f(n,m) : \text{существуют положительные константы } c, n_0 \text{ и } m_0, \text{ такие, что } 0 \leq f(n,m) \leq cg(n,m) \text{ для всех } n \geq n_0 \text{ или } m \geq m_0\} .$$

Приведите соответствующие определения для $\Omega(g(n,m))$ и $\Theta(g(n,m))$.

3.2. Стандартные обозначения и часто встречающиеся функции

В этом разделе рассматриваются некоторые стандартные математические функции и обозначения, а также исследуются взаимоотношения между ними. В нем, кроме того, иллюстрируется применение асимптотических обозначений.

Монотонность

Функция $f(n)$ является ***монотонно неубывающей*** (monotonically increasing), если из $m \leq n$ вытекает $f(m) \leq f(n)$. Аналогично она является ***монотонно невозрастающей*** (monotonically decreasing), если из $m \leq n$ вытекает $f(m) \geq f(n)$. Функция $f(n)$ является ***монотонно возрастающей*** (strictly increasing), если из $m < n$ вытекает $f(m) < f(n)$, и ***монотонно убывающей*** (strictly decreasing), если из $m < n$ вытекает $f(m) > f(n)$.

Полы и потолки

Для любого действительного числа x обозначим наибольшее целое число, меньшее или равное x, как $\lfloor x \rfloor$ (читается как "пол (floor) x"), а наименьшее целое число, большее или равное x, — как $\lceil x \rceil$ (читается как "потолок (ceil) x"). Для всех действительных x

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 . \tag{3.3}$$

Для любого целого n

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n ,$$

а для любого действительного числа $x \ge 0$ и целых чисел $a, b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil , \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor , \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \le \frac{a + (b-1)}{b} , \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ge \frac{a - (b-1)}{b} . \tag{3.7}$$

Функция $f(x) = \lfloor x \rfloor$ является монотонно неубывающей, как и функция $f(x) = \lceil x \rceil$.

Модульная арифметика

Для любого целого числа a и любого натурального n величина $a \bmod n$ представляет собой ***остаток*** от деления a на n:

$$a \bmod n = a - n \lfloor a/n \rfloor . \tag{3.8}$$

Отсюда следует, что

$$0 \le a \bmod n < n . \tag{3.9}$$

Располагая подобным определением, удобно ввести специальные обозначения для указания того, что два целых числа имеют одинаковые остатки при делении на какое-то натуральное число. Тот факт, что $(a \bmod n) = (b \bmod n)$, записывается как $a \equiv b \pmod n$; при этом говорят, что число a ***эквивалентно***, или ***равно***, числу b по модулю n (или что числа a и b сравнимы по модулю n). Другими словами, $a \equiv b \pmod n$, если числа a и b дают одинаковые остатки при делении на n. Это эквивалентно утверждению, что $a \equiv b \pmod n$ тогда и только тогда, когда n является делителем числа $b-a$. Запись $a \not\equiv b \pmod n$ означает, что число a не эквивалентно числу b по модулю n.

Полиномы

Для заданного неотрицательного целого d полиномом степени d от аргумента n называется функция $p(n)$ вида

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

где константы $a_0, a_1, \ldots, a_d$ — ***коэффициенты*** полинома и $a_d \ne 0$. Полином является асимптотически положительной функцией тогда и только тогда, когда $a_d > 0$. Для асимптотически положительных полиномов $p(n)$ степени d справедливо соотношение $p(n) = \Theta(n^d)$. Для любой действительной константы $a \ge 0$ функция n^a монотонно неубывающая, а для $a \le 0$ эта функция монотонно невозраста-

ющая. Говорят, что функция $f(n)$ ***полиномиально ограничена***, если существует такая константа k, что $f(n) = O(n^k)$.

Показательные функции

Для всех действительных чисел $a > 0$, m и n справедливы следующие тождества.

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= 1/a \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

Для всех n и $a \geq 1$ функция a^n является монотонно неубывающей функцией аргумента n. Для удобства будем считать, что $0^0 = 1$.

Соотношение между скоростями роста полиномов и показательных функций можно определить исходя из того факта, что для любых действительных констант a и b, таких, что $a > 1$,

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 , \tag{3.10}$$

откуда можно заключить, что

$$n^b = o(a^n) .$$

Таким образом, любая показательная функция, основание a которой строго больше единицы, возрастает быстрее любой полиномиальной функции.

Обозначив через e основание натурального логарифма (приблизительно равное 2.718281828...), можем записать следующее соотношение, справедливое для любого действительного x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{3.11}$$

где "!" обозначает факториал, определенный ниже в этом разделе. Для всех действительных x справедливо следующее неравенство:

$$e^x \geq 1 + x , \tag{3.12}$$

где равенство выполняется только при $x = 0$. Когда $|x| \leq 1$, можно использовать такое приближение:

$$1 + x \leq e^x \leq 1 + x + x^2 . \tag{3.13}$$

При $x \to 0$ приближение e^x функцией $1 + x$ вполне удовлетворительно:

$$e^x = 1 + x + \Theta(x^2) .$$

В этом уравнении асимптотические обозначения используются для описания предельного поведения при $x \to 0$, а не при $x \to \infty$. Для всех x мы имеем

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x \ . \tag{3.14}$$

Логарифмы

Мы будем использовать следующие обозначения.

$\lg n = \log_2 n$	(бинарный логарифм)
$\ln n = \log_e n$	(натуральный логарифм)
$\lg^k n = (\lg n)^k$	(возведение в степень)
$\lg\lg n = \lg(\lg n)$	(композиция)

Важное соглашение, которое мы приняли в книге, — *логарифмические функции применяются только к ближайшему члену выражения*. Например, $\lg n + k$ означает $(\lg n) + k$, а не $\lg(n + k)$. Если основание логарифма $b > 1$, то при $n > 0$ функция $\log_b n$ монотонно возрастает.

Для всех действительных $a > 0$, $b > 0$, $c > 0$ и n

$$\begin{aligned}
a &= b^{\log_b a} \ , \\
\log_c(ab) &= \log_c a + \log_c b \ , \\
\log_b a^n &= n \log_b a \ , \\
\log_b a &= \frac{\log_c a}{\log_c b} \ , \qquad (3.15) \\
\log_b(1/a) &= -\log_b a \ , \\
\log_b a &= \frac{1}{\log_a b} \ , \\
a^{\log_b c} &= c^{\log_b a} \ , \qquad (3.16)
\end{aligned}$$

где в каждом из приведенных уравнений основание логарифма не равно 1.

Согласно уравнению (3.15) изменение основания логарифма приводит к умножению значения этого логарифма только на постоянный множитель, поэтому мы часто будем использовать обозначение "$\lg n$", не заботясь о постоянном множителе, как это делается в O-обозначениях. Специалисты в области вычислительной техники считают наиболее естественной основой логарифма число 2, так как во многих алгоритмах и структурах данных производится разбиение задачи на две части.

При $|x| < 1$ имеется простое разложение в ряд

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots \ .$$

Кроме того, для $x > -1$ выполняются следующие неравенства:

$$\frac{x}{1+x} \le \ln(1+x) \le x \ , \tag{3.17}$$

где равенство достигается только при $x = 0$.

Говорят, что функция $f(n)$ ***полилогарифмически ограничена***, если существует такая константа k, что $f(n) = O(\lg^k n)$. Соотношение между скоростью роста полиномов и полилогарифмов можно найти, подставив в уравнение (3.10) $\lg n$ вместо n и 2^a вместо a, в результате чего получим

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 \ .$$

Из приведенного выше соотношения можно заключить, что для произвольной константы $a > 0$

$$\lg^b n = o(n^a) \ .$$

Таким образом, любая положительная полиномиальная функция возрастает быстрее, чем любая полилогарифмическая функция.

Факториалы

Обозначение $n!$ (читается как "n факториал") определено для целых чисел $n \ge 0$ следующим образом:

$$n! = \begin{cases} 1 \ , & \text{если } n = 0 \ , \\ n \cdot (n-1)! \ , & \text{если } n > 0 \ . \end{cases}$$

Таким образом, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Слабой верхней границей факториала является $n! \le n^n$, поскольку каждый из n членов, входящих в факториальное произведение, не превышает n. ***Формула Стирлинга***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \ , \tag{3.18}$$

где e — основание натуральных логарифмов, дает более точную верхнюю (а также нижнюю) границу. В упр. 3.2.3 требуется доказать, что

$$n! = o(n^n) \ ,$$
$$n! = \omega(2^n) \ ,$$
$$\lg(n!) = \Theta(n \lg n) \ , \tag{3.19}$$

причем при доказательстве уравнения (3.19) удобно использовать формулу Стирлинга. Для всех $n \ge 1$ справедливо также следующее равенство:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \ , \tag{3.20}$$

где

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} . \tag{3.21}$$

Функциональная итерация

Запись $f^{(i)}(n)$ используется для обозначения функции $f(n)$, итеративно примененной i раз к исходному значению n. Подходя формально, пусть $f(n)$ является функцией от действительного аргумента. Для неотрицательных целых i рекурсивно определим

$$f^{(i)}(n) = \begin{cases} n \, , & \text{если } i = 0 \, , \\ f(f^{(i-1)}(n)) \, , & \text{если } i > 0 \, . \end{cases}$$

Например, если $f(n) = 2n$, то $f^{(i)}(n) = 2^i n$.

Итерированная логарифмическая функция

Обозначение $\lg^* n$ (читается как "логарифм со звездочкой от n") будет применяться для указания повторно применяемого логарифма, который определяется следующим образом. Пусть $\lg^{(i)} n$ представляет собой итерацию функции $f(n) = \lg n$. Поскольку логарифм от отрицательных чисел не определен, функция $\lg^{(i)} n$ определена, только если $\lg^{(i-1)} n > 0$. Не перепутайте обозначения $\lg^{(i)} n$ (логарифм, примененный последовательно i раз к аргументу n) и $\lg^i n$ (логарифм n, возведенный в i-ю степень). Итерированный логарифм определяется следующим образом:

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\} .$$

Это *очень* медленно растущая функция.

$$\begin{aligned} \lg^* 2 &= 1 \\ \lg^* 4 &= 2 \\ \lg^* 16 &= 3 \\ \lg^* 65536 &= 4 \\ \lg^*(2^{65536}) &= 5 \end{aligned}$$

Поскольку количество атомов в видимой Вселенной оценивается примерно в 10^{80}, что гораздо меньше 2^{65536}, мы вряд ли встретимся с входными данными размером n, таким, что $\lg^* n > 5$.

Числа Фибоначчи

Числа Фибоначчи определяются с помощью следующего рекуррентного соотношения.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{для } i \geq 2 \end{aligned} \tag{3.22}$$

Таким образом, каждое число Фибоначчи представляет собой сумму двух предыдущих, что дает нам последовательность

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots\ .$$

Числа Фибоначчи тесно связаны с ***золотым сечением*** ϕ и сопряженным с ним значением $\widehat{\phi}$, которые определяются уравнением

$$x^2 = x + 1\ , \tag{3.23}$$

и задаются следующими формулами (см. упр. 3.2.6).

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -0.61803\ldots \end{aligned} \tag{3.24}$$

В частности, мы имеем

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}\ ,$$

что можно доказать по индукции (упр. 3.2.7). Поскольку $\left|\widehat{\phi}\right| < 1$, мы имеем

$$\begin{aligned} \frac{\left|\widehat{\phi}^i\right|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2}\ , \end{aligned}$$

откуда вытекает, что

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor\ , \tag{3.25}$$

что в переводе на обычный язык гласит: "i-е число Фибоначчи F_i равно $\phi^i/\sqrt{5}$, округленного до ближайшего целого числа". Таким образом, числа Фибоначчи растут экспоненциально.

Упражнения

3.2.1

Покажите, что если функции $f(n)$ и $g(n)$ монотонно неубывающие, то таковыми же являются и функции $f(n) + g(n)$ и $f(g(n))$, а если вдобавок $f(n)$ и $g(n)$ неотрицательны, то монотонно неубывающей является и функция $f(n) \cdot g(n)$.

3.2.2

Докажите уравнение (3.16).

3.2.3

Докажите уравнение (3.19). Докажите также, что $n! = \omega(2^n)$ и $n! = o(n^n)$.

3.2.4 ★

Является ли функция $\lceil \lg n \rceil!$ полиномиально ограниченной? А функция $\lceil \lg \lg n \rceil!$?

3.2.5 ★

Какая из функций $\lg(\lg^* n)$ и $\lg^*(\lg n)$ является асимптотически большей?

3.2.6

Покажите, что золотое сечение ϕ и сопряженное с ним $\widehat{\phi}$ удовлетворяют уравнению $x^2 = x + 1$.

3.2.7

Докажите по индукции, что i-е число Фибоначчи удовлетворяет уравнению

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} ,$$

где ϕ — золотое сечение, а $\widehat{\phi}$ — сопряженное с ним.

3.2.8

Покажите, что из $k \ln k = \Theta(n)$ вытекает $k = \Theta(n/\ln n)$.

Задачи

3.1. Асимптотическое поведение полиномов

Пусть

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

где $a_d > 0$, представляет собой полином степени d от n, и пусть k является константой. Используя определения асимптотических обозначений, докажите следующие свойства.

а. Если $k \geq d$, то $p(n) = O(n^k)$.

б. Если $k \le d$, то $p(n) = \Omega(n^k)$.

в. Если $k = d$, то $p(n) = \Theta(n^k)$.

г. Если $k > d$, то $p(n) = o(n^k)$.

д. Если $k < d$, то $p(n) = \omega(n^k)$.

3.2. Относительный асимптотический рост

Для каждой пары приведенных в таблице выражений (A, B) укажите, каким отношением A связано с B: O, o, Ω, ω или Θ. Предполагается, что $k \ge 1$, $\epsilon > 0$ и $c > 1$ — константы. Ваш ответ должен выражаться таблицей, в каждой ячейке которой указано значение "Да" или "Нет".

	A	B	O	o	Ω	ω	Θ
а.	$\lg^k n$	n^ϵ					
б.	n^k	c^n					
в.	$\sqrt{n}$	$n^{\sin n}$					
г.	2^n	$2^{n/2}$					
д.	$n^{\lg c}$	$c^{\lg n}$					
е.	$\lg(n!)$	$\lg(n^n)$					

3.3. Упорядочение по скорости асимптотического роста

а. Расположите приведенные ниже функции по скорости их асимптотического роста, т.е. постройте такую последовательность функций $g_1, g_2, \ldots, g_{30}$, что $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$. Разбейте свой список на классы эквивалентности так, чтобы функции $f(n)$ и $g(n)$ находились в одном и том же классе тогда и только тогда, когда $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

б. Приведите пример неотрицательной функции $f(n)$, такой, что для всех функций $g_i(n)$ из части (а) $f(n)$ не принадлежит ни множеству $O(g_i(n))$, ни множеству $\Omega(g_i(n))$.

3.4. Свойства асимптотических обозначений

Пусть $f(n)$ и $g(n)$ — асимптотически положительные функции. Докажите или опровергните справедливость каждого из приведенных ниже утверждений.

а. Из $f(n) = O(g(n))$ вытекает $g(n) = O(f(n))$.

б. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

в. Из $f(n) = O(g(n))$ вытекает $\lg(f(n)) = O(\lg(g(n)))$, где $\lg(g(n)) \geq 1$ и $f(n) \geq 1$ для всех достаточно больших n.

г. Из $f(n) = O(g(n))$ вытекает $2^{f(n)} = O\left(2^{g(n)}\right)$.

д. $f(n) = O\left((f(n))^2\right)$.

е. Из $f(n) = O(g(n))$ вытекает $g(n) = \Omega(f(n))$.

ж. $f(n) = \Theta(f(n/2))$.

з. $f(n) + o(f(n)) = \Theta(f(n))$.

3.5. Вариации определений O и Ω

Некоторые авторы определяют Ω несколько иначе, чем это делали мы; давайте для такого альтернативного определения будем использовать символ $\overset{\infty}{\Omega}$ (читается как “омега бесконечность”). Будем говорить, что $f(n) = \overset{\infty}{\Omega}(g(n))$, если существует положительная константа c, такая, что $f(n) \geq cg(n) \geq 0$ для бесконечного количества целых чисел n.

а. Покажите, что для любых двух асимптотически неотрицательных функций $f(n)$ и $g(n)$ выполняется одно из соотношений $f(n) = O(g(n))$ и $f(n) = \overset{\infty}{\Omega}(g(n))$ (или они оба), в то время как при использовании Ω вместо $\overset{\infty}{\Omega}$ это утверждение ложно.

б. Опишите потенциальные преимущества и недостатки применения $\overset{\infty}{\Omega}$ вместо Ω для характеристики времени работы программ.

Некоторые авторы определяют несколько иначе и O; для такого альтернативного определения будем использовать символ O'. Будем говорить, что $f(n) = O'(g(n))$ тогда и только тогда, когда $|f(n)| = O(g(n))$.

в. Что произойдет в теореме 3.1 с каждым из направлений “тогда и только тогда, когда”, если заменить O на O', но оставить без изменений Ω?

Некоторые авторы определяют $\widetilde{O}$ (читается как “о с тильдой”) для обозначения O, в котором игнорируются логарифмические множители:

$$\widetilde{O}(g(n)) = \{f(n) : \text{существуют положительные константы } c,\ k \text{ и } n_0, \text{ такие, что } 0 \leq f(n) \leq cg(n)\lg^k(n) \text{ для всех } n \geq n_0\} .$$

г. Определите $\widetilde{\Omega}$ и $\widetilde{\Theta}$ аналогичным образом. Докажите соответствующий аналог теоремы 3.1.

3.6. Итерированные функции

Оператор итерации $*$, используемый в функции $\lg^*$, можно применить к любой монотонно возрастающей функции $f(n)$, определенной на множестве действительных чисел. Для заданной константы $c \in \mathbb{R}$ итерированная функция f_c^* определяется следующим образом:

$$f_c^*(n) = \min\left\{i \geq 0 : f^{(i)}(n) \leq c\right\} .$$

Она может не быть вполне определенной во всех случаях. Другими словами, величина $f_c^*(n)$ представляет собой количество итерированных применений функции f, требующееся для того, чтобы уменьшить ее аргумент до значения, не превышающего c.

Для каждой из приведенных далее функций $f(n)$ и констант c дайте максимально точную оценку функции $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
а.	$n-1$	0	
б.	$\lg n$	1	
в.	$n/2$	1	
г.	$n/2$	2	
д.	$\sqrt{n}$	2	
е.	$\sqrt{n}$	1	
ж.	$n^{1/3}$	2	
з.	$n/\lg n$	2	

Заключительные замечания

Кнут (Knuth) [208][3] попытался выяснить происхождение O-обозначений и обнаружил, что впервые они появились в 1892 году в учебнике П. Бахманна (P. Bachmann) по теории чисел. o-обозначения были введены в 1909 году Э. Ландау (E. Landau) при обсуждении распределения простых чисел. Введение Ω- и Θ-обозначений приписывают Кнуту [212], который исправил неточность популярного в литературе, но технически неаккуратного применения O-обозначений как для верхней, так и для нижней асимптотических границ. Многие продолжают использовать O-обозначения там, где более уместны были бы Θ-обозначения. Дальнейшее обсуждение исторического развития асимптотических обозначений можно найти у Кнута [208, 212], а также Брассарда (Brassard) и Брейтли (Bratley) [53].

Определения асимптотических обозначений у разных авторов иногда различаются, однако в большинстве часто встречающихся ситуаций они дают согласу-

[3]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

ющиеся результаты. В некоторых альтернативных случаях подразумевается, что функции не являются асимптотически неотрицательными, поэтому ограничению подлежит их абсолютное значение.

Равенство (3.20) было получено Роббинсом (Robbins) [295]. Другие свойства элементарных математических функций можно найти в любом хорошем справочнике по математике, например, в справочнике Абрамовича (Abramowitch) и Стеган (Stegun) [1] или Цвиллингера (Zwillinger) [360], а также в книгах по вычислительной математике, таких как книги Апостола (Apostol) [18] или Томаса (Thomas) и др. [332]. В книгах Кнута [208], а также Грехема (Graham), Кнута и Паташника (Patashnik) [151][4] содержится множество материала по дискретной математике, который используется в информатике.

[4]Имеется русский перевод: Р. Грэхем, Д. Кнут, О. Паташник. *Конкретная математика. Математические основы информатики, 2-е изд.* — М.: И.Д. "Вильямс", 2010.

Глава 4. Разделяй и властвуй

В разделе 2.3.1 вы познакомились с применением парадигмы "разделяй и властвуй" на примере сортировки слиянием. Напомним, что при данном подходе мы решаем задачу рекурсивно, применяя на каждом уровне рекурсии три шага.

Разделение задачи на несколько подзадач, которые представляют собой меньшие экземпляры той же задачи.

Властвование над подзадачами путем их рекурсивного решения. Если размеры подзадач достаточно малы, такие подзадачи могут решаться непосредственно.

Комбинирование решений подзадач в решение исходной задачи.

Если подзадачи достаточно велики для рекурсивного решения, мы называем эту ситуацию ***рекурсивным случаем***. Если подзадачи становятся достаточно малы для того, чтобы не прибегать к рекурсии, мы говорим, что рекурсия "достигает дна" и опускается до ***базового случая***. Иногда, в дополнение к подзадачам, которые представляют собой меньшие экземпляры той же задачи, приходится решать подзадачи, несколько отличающиеся от исходной задачи. Мы рассматриваем решение таких подзадач как часть шага комбинирования.

В этой главе мы рассмотрим новые алгоритмы, основанные на подходе "разделяй и властвуй". Первый из них решает задачу максимального подмассива: на вход алгоритма поступает массив чисел, и необходимо найти непрерывный подмассив, значения которого дают наибольшую сумму. Затем мы рассмотрим два алгоритма "разделяй и властвуй", предназначенные для умножения матриц размером $n \times n$. Один из них имеет время работы $\Theta(n^3)$, что не лучше, чем время работы обычного алгоритма непосредственного умножения, но второй, алгоритм Штрассена, выполняет умножение за время $O(n^{2.81})$, так что он асимптотически лучше, чем алгоритм непосредственного умножения.

Рекуррентные соотношения

Рука об руку с парадигмой "разделяй и властвуй" идут рекуррентные соотношения, поскольку они предоставляют естественный способ описания времени работы соответствующих алгоритмов. ***Рекуррентное соотношение*** представляет собой уравнение или неравенство, которое описывает функцию через ее значения

для меньших аргументов. Например, в разделе 2.3.2 мы описывали время $T(n)$ работы процедуры MERGE-SORT в наихудшем случае с помощью рекуррентного соотношения

$$T(n) = \begin{cases} \Theta(1) , & \text{если } n = 1 , \\ 2T(n/2) + \Theta(n) , & \text{если } n > 1 , \end{cases} \tag{4.1}$$

решением которого является функция $T(n) = \Theta(n \lg n)$.

Рекуррентные соотношения могут принимать множество разных форм. Например, рекурсивный алгоритм может делить задачу на подзадачи разного размера, например, разбивая на части, представляющие собой $2/3$ и $1/3$ исходной задачи. Если при этом разделение и комбинирование выполняются за линейное время, время работы такого алгоритма определяется рекуррентным соотношением $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Подзадачи не обязаны быть ограниченными некоторыми постоянными долями размера исходной задачи. Например, рекурсивная версия линейного поиска (см. упр. 2.1.3) создает только одну подзадачу, содержащую только на один элемент меньше, чем исходная задача. Каждый рекурсивный вызов требует константного времени плюс время на рекурсивный вызов, в свою очередь осуществляемый им, что приводит к рекуррентному соотношению $T(n) = T(n-1) + \Theta(1)$.

В этой главе предлагаются три метода решения рекуррентных соотношений, т.е. получения асимптотических границ решения "Θ" или "O".

- В ***методе подстановки*** мы делаем предположение о границе, а затем используем метод математической индукции для доказательства его корректности.
- В ***методе деревьев рекурсии*** рекуррентное соотношение преобразуется в дерево, узлы которого представляют стоимости на разных уровнях рекурсии. Затем для решения рекуррентного соотношения используются методы оценки сумм.
- В ***основном методе*** (master method) граничные оценки рекуррентных соотношений представляются в виде

$$T(n) = aT(n/b) + f(n) , \tag{4.2}$$

где $a \geq 1$, $b > 1$, а $f(n)$ — заданная функция. Такие рекуррентные соотношения возникают очень часто. Рекуррентное соотношение вида (4.2) описывает алгоритм "разделяй и властвуй", который создает a подзадач, каждая из которых имеет размер, равный $1/b$ размера исходной задачи, и шаги разделения и комбинирования которого в сумме занимают время $f(n)$.

Для использования основного метода вам нужно запомнить три разных случая, после чего вы сможете легко определять асимптотические границы для многих простых рекуррентных соотношений. Мы будем использовать данный метод для определения времени работы алгоритмов "разделяй и властвуй" для поиска максимального подмассива и для умножения матриц, а также для множества алгоритмов, основанных на этой парадигме, в других местах книги.

Иногда нам будут встречаться рекуррентные соотношения, которые представляют собой неравенства, такие как $T(n) \leq 2T(n/2) + \Theta(n)$. Поскольку такие рекуррентные соотношения указывают только верхнюю границу $T(n)$, их решения мы будем записывать с применением O-обозначений (а не Θ-обозначений). Аналогично, если это неравенство обратить (так, что оно примет вид $T(n) \geq 2T(n/2)+\Theta(n)$), то, поскольку такое рекуррентное соотношение дает только нижнюю границу $T(n)$, в его решении будут использоваться Ω-обозначения.

Технические детали рекуррентных соотношений

На практике при составлении и решении рекуррентных соотношений мы пренебрегаем рядом технических деталей. Например, если мы вызываем процедуру MERGE-SORT для нечетного количества элементов n, то в результате мы получаем подзадачи размером $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil$. Ни один из этих размеров в действительности не равен $n/2$, поскольку $n/2$ не является целым числом при нечетном n. Технически рекуррентное соотношение, описывающее время работы процедуры MERGE-SORT в наихудшем случае, в действительности равно

$$T(n) = \begin{cases} \Theta(1)\ , & \text{если } n = 1\ , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)\ , & \text{если } n > 1\ . \end{cases} \tag{4.3}$$

Граничные условия — еще один пример технических особенностей, которые обычно игнорируются. Поскольку время работы алгоритма с входными данными фиксированного размера выражается константой, то в рекуррентных соотношениях, описывающих время работы алгоритмов, для достаточно малых n обычно справедливо соотношение $T(n) = \Theta(1)$. Поэтому для удобства граничные условия рекуррентных соотношений, как правило, опускаются и предполагается, что для малых n время работы алгоритма $T(n)$ является константой. Например, рекуррентное соотношение (4.1) обычно записывается как

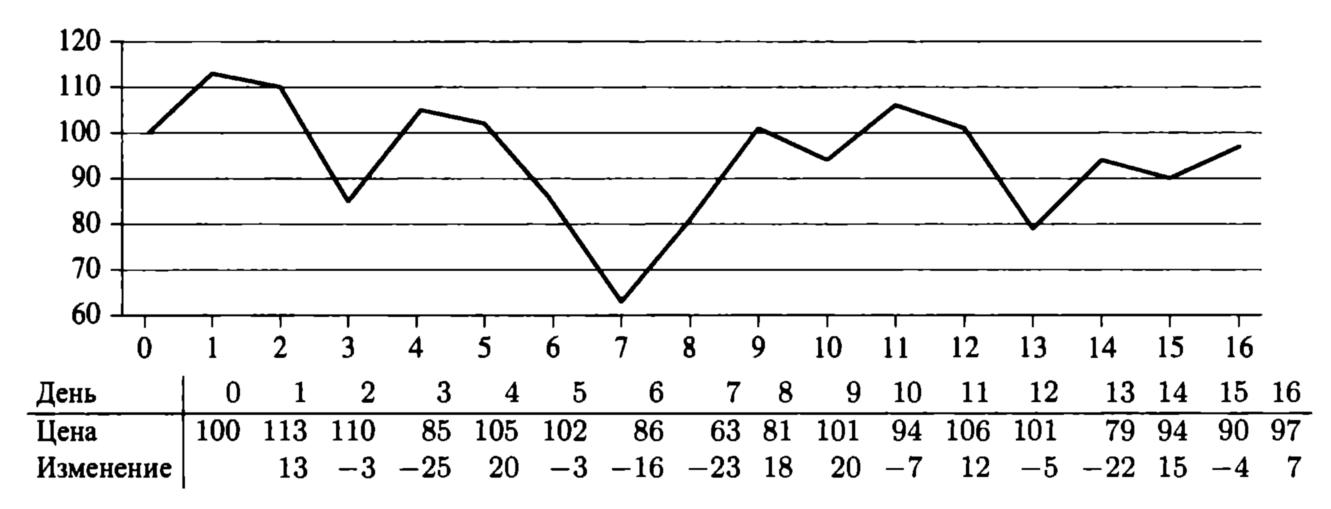
$$T(n) = 2T(n/2) + \Theta(n)\ , \tag{4.4}$$

без явного указания значений $T(n)$ для малых n. Причина состоит в том, что хотя изменение значения $T(1)$ и приводит к изменению решения рекуррентного соотношения, это решение обычно изменяется не более чем на постоянный множитель, так что порядок роста остается неизменным.

Формулируя и решая рекуррентные соотношения, мы часто опускаем полы, потолки и граничные условия. Мы продвигаемся вперед без этих деталей, а затем выясняем, важны они или нет. Обычно они не играют никакой роли, но мы должны знать, когда это не так. В подобных ситуациях помогают опыт, а также некоторые теоремы. В этих теоремах формулируются условия, когда упомянутые детали не влияют на асимптотическое поведение рекуррентных соотношений, возникающих в ходе анализа алгоритмов (см. теорему 4.1). Однако в данной главе мы не будем опускать технические детали, так как это позволит продемонстрировать некоторые тонкости, присущие методам решения рекуррентных соотношений.

4.1. Задача поиска максимального подмассива

Предположим, что у вас появилась возможность вложить деньги в корпорацию по производству неустойчивых химических соединений Volatile Chemical Corporation. Подобно производимой продукции цена акций Volatile Chemical Corporation тоже очень неустойчива. Вы можете купить только один комплект акций и продать его в какой-то другой день, осуществляя покупки и продажи после закрытия торгов. Ваши неудобства компенсируются тем, что вы владеете информацией о будущих ценах на акции. Ваша цель — получить максимальную прибыль. На рис. 4.1 показана цена акций за 17-дневный период. Вы можете покупать акции в любой день, начиная с нулевого дня, когда цена равна $100. Конечно, вы захотите купить подешевле, а продать подороже, но — увы! — так может и не получиться. На рис. 4.1 наименьшая цена достигается после седьмого дня, т.е. после того, как будет достигнута максимальная цена после первого дня.



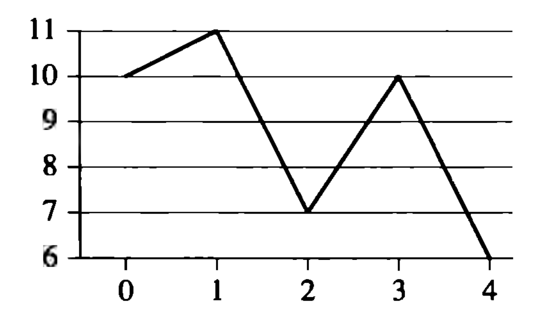
День	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Цена	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Изменение		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Рис. 4.1. Информация о цене акций Volatile Chemical Corporation после закрытия торгов за 17-дневный период. На горизонтальной оси диаграммы указан день торгов, на вертикальной — цена. В нижней строке таблицы указано изменение цены по сравнению с предыдущим днем.

Можно решить, что прибыль всегда можно максимизировать, либо покупая по наименьшей цене, либо продавая по наибольшей. Например, на рис. 4.1 можно было бы максимизировать прибыль, выполняя покупку по наименьшей цене, которая достигается после торгов седьмого дня. Если бы такая стратегия всегда работала, то было бы просто определить, как максимизировать прибыль: найти наибольшую и наименьшую цены, а затем слева от наивысшей цены выполнить поиск наинизшей, а справа от наинизшей цены выполнить поиск наивысшей и взять пару с максимальным отличием. Но на рис. 4.2 показан контрпример, демонстрирующий, что иногда максимальная прибыль достигается и не при покупке по наинизшей цене, и не при продаже по наивысшей.

Перебор

Можно легко разработать решение этой задачи, основанное на грубой силе: просто испытать все возможные пары дат покупки и продажи, в которых дата



День	0	1	2	3	4
Цена	10	11	7	10	6
Изменение		1	−4	3	−4

Рис. 4.2. Пример, демонстрирующий, что максимальная прибыль не всегда начинается с минимальной цены и не всегда заканчивается максимальной. Здесь вновь на горизонтальной оси диаграммы указан день торгов, на вертикальной — цена. Максимальная прибыль в \$3 достигается при покупке после торгов во второй день и продаже после торгов в третий день. Но цена \$7 во второй день не является наинизшей, как и цена \$10 в третий день не является наивысшей.

покупки предшествует дате продажи. Период из n дней имеет $\binom{n}{2}$ таких пар дат. Поскольку $\binom{n}{2}$ представляет собой $\Theta(n^2)$, и лучшее, на что мы можем надеяться, — это вычисление каждой пары за константное время, такой подход будет давать время работы $\Omega(n^2)$. Нельзя ли достичь чего-то лучшего?

Преобразование

Чтобы получить алгоритм со временем работы $o(n^2)$, взглянем на входные данные под несколько иным углом. Мы хотим найти последовательность дней, для которых итоговая разница между первым и последним днем максимальна. Вместо того чтобы работать с ежедневными ценами, давайте рассмотрим ежедневное изменение цены, где изменение в день i представляет собой разность между ценой после торгов в день i и ценой в день $i - 1$. На рис. 4.1 эти изменения показаны в нижней строке. Если рассматривать эту строку как массив A, показанный на рис. 4.3, то задача заключается в поиске непустого непрерывного подмассива A, значения которого имеют наибольшую сумму. Назовем такой непрерывный подмассив ***максимальным подмассивом***. Например, в массиве на рис. 4.3 максимальным подмассивом массива $A[1..16]$ является $A[8..11]$ с суммой 43. Таким образом, лучше всего покупать акции непосредственно перед восьмым днем (т.е. после торгов седьмого дня), а продавать после торгов одиннадцатого дня, получая при этом прибыль в \$43 с акции.

На первый взгляд, такое преобразованием ничем не может нам помочь. Нам все равно нужно проверить $\binom{n-1}{2} = \Theta(n^2)$ подмассивов в случае периода из n дней. В упр. 4.1.2 требуется показать, что хотя вычисление стоимости одного подмассива может потребовать времени, пропорционального длине этого подмас-

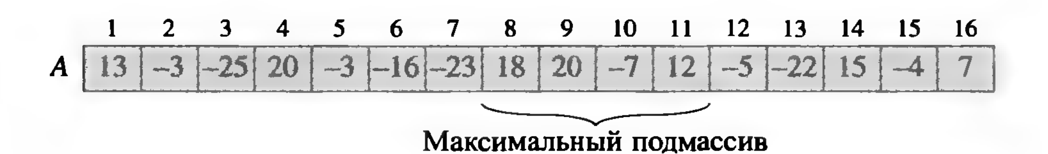


Рис. 4.3. Изменения цен как входные данные для задачи поиска максимального подмассива. Здесь подмассив $A[8..11]$ с суммой 43 имеет наибольшую сумму среди всех непрерывных подмассивов массива A.

сива, при вычислении всех $\Theta(n^2)$ сумм подмассивов можно организовать вычисления таким образом, что каждый подмассив с учетом уже вычисленных ранее сумм подмассивов будет вычисляться за время $O(1)$. Таким образом, решение методом грубой силы потребует $\Theta(n^2)$ времени.

Так что давайте поищем более эффективное решение задачи максимального подмассива. Говоря о максимальном подмассиве в единственном числе, следует отдавать себе отчет в том, что таких подмассивов, сумма элементов которых достигает максимального значения, может быть несколько.

Задача поиска максимального подмассива интересна только в том случае, когда массив содержит некоторое количество отрицательных значений. Если же все элементы массива положительны, то задача поиска максимального подмассива становится тривиальной, так как в этом случае наибольшую сумму имеет весь массив целиком.

Решение "разделяй и властвуй"

Давайте подумаем о том, как можно решить задачу поиска максимального подмассива с использованием метода "разделяй и властвуй". Предположим, что мы хотим найти максимальный подмассив подмассива $A[low \mathinner{..} high]$. Технология "разделяй и властвуй" предполагает, что мы делим массив на две части, по возможности одинакового размера, т.е. мы находим среднюю точку подмассива, скажем, mid, и рассматриваем подмассивы $A[low \mathinner{..} mid]$ и $A[mid + 1 \mathinner{..} high]$. Как показано на рис. 4.4, (а), любой непрерывный подмассив $A[i \mathinner{..} j]$ массива $A[low \mathinner{..} high]$ должен находиться только в одном из следующих положений:

- полностью располагаться в подмассиве $A[low \mathinner{..} mid]$, так что $low \le i \le j \le mid$;
- полностью располагаться в подмассиве $A[mid + 1 \mathinner{..} high]$, так что $mid < i \le j \le high$;
- пересекать среднюю точку, так что $low \le i \le mid < j \le high$.

Следовательно, максимальный подмассив массива $A[low \mathinner{..} high]$ должен располагаться ровно одним из этих способов. Фактически максимальный подмассив массива $A[low \mathinner{..} high]$ должен иметь наибольшую сумму среди всех

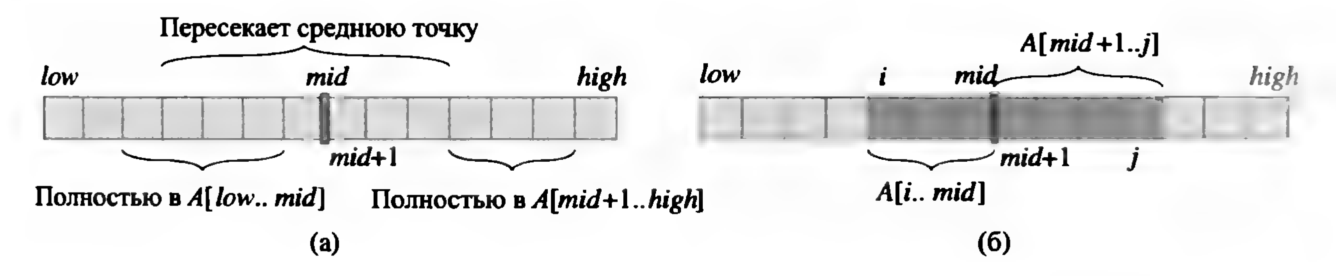


Рис. 4.4. (а) Возможные размещения подмассивов массива $A[low \mathinner{..} high]$: полностью в $A[low \mathinner{..} mid]$, полностью в $A[mid + 1 \mathinner{..} high]$ или с пересечением средней точки mid. **(б)** Любой подмассив массива $A[low \mathinner{..} high]$, пересекающий среднюю точку, содержит два подмассива $A[i \mathinner{..} mid]$ и $A[mid + 1 \mathinner{..} j]$, где $low \le i \le mid$ и $mid < j \le high$.

подмассивов, полностью находящихся в $A[low \,..\, mid]$, полностью находящихся в $A[mid+1 \,..\, high]$ или пересекающих среднюю точку. Максимальные подмассивы $A[low \,..\, mid]$ и $A[mid+1 \,..\, high]$ можно найти рекурсивно, поскольку эти две подзадачи представляют собой экземпляры задачи поиска максимального подмассива меньшего размера. Таким образом, все, что осталось сделать, — это найти максимальный подмассив, пересекающий среднюю точку, и выбрать из этих трех подмассивов тот, у которого будет наибольшая сумма.

Можно легко найти максимальный подмассив, пересекающий среднюю точку, за время, линейно зависящее от размера подмассива $A[low \,..\, high]$. Эта задача *не* является меньшим экземпляром нашей исходной задачи, поскольку при этом добавлено ограничение, что подмассив должен пересекать среднюю точку. Как показано на рис. 4.4, (б), любой подмассив, пересекающий среднюю точку, состоит из двух подмассивов $A[i \,..\, mid]$ и $A[mid+1 \,..\, j]$, где $low \le i \le mid$ и $mid < j \le high$. Следовательно, нужно просто найти максимальные подмассивы вида $A[i \,..\, mid]$ и $A[mid+1 \,..\, j]$, а затем объединить их. Процедура FIND-MAX-CROSSING-SUBARRAY получает в качестве входных данных массив A и индексы low, mid и $high$, и возвращает кортеж, содержащий индексы, определяющие пересекающий среднюю точку максимальный подмассив, а также сумму значений элементов этого максимального подмассива.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = -∞
 2  sum = 0
 3  for i = mid downto low
 4      sum = sum + A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = -∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Эта процедура работает следующим образом. Строки 1–7 находят максимальный подмассив в левой половине, $A[low \,..\, mid]$. Поскольку этот подмассив должен содержать $A[mid]$, цикл **for** в строках 3–7 начинает работу со значения индекса i, равного mid, и идет вниз до значения low, так что каждый рассматриваемый подмассив имеет вид $A[i \,..\, mid]$. В строках 1 и 2 выполняется инициализация переменной *left-sum*, в которой хранится наибольшая найденная к этому моменту сумма, и *sum*, хранящей сумму элементов в $A[i \,..\, mid]$. Когда в строке 5 мы находим подмассив $A[i \,..\, mid]$ с суммой значений, большей, чем *left-sum*, мы об-

новляем значение переменной *left-sum*, делая его равным этой сумме в строке 6, а в строке 7 мы обновляем переменную *max-left*, записывая в нее индекс i. Строки 8–14 аналогично работают в правой половине, $A[mid{+}1\,..\,high]$. Здесь цикл **for** в строках 10–14 начинает работу со значения индекса j, равного $mid + 1$, и идет вверх до значения *high*, так что каждый рассматриваемый подмассив имеет вид $A[mid{+}1\,..\,j]$. Наконец строка 15 возвращает индексы *max-left* и *max-right*, которые определяют максимальный подмассив, пересекающий среднюю точку, вместе с суммой *left-sum* + *right-sum* значений в подмассиве $A[max\text{-}left\,..\,max\text{-}right]$.

Если подмассив $A[low\,..\,high]$ содержит n элементов (так что $n = high - low + 1$), мы утверждаем, что вызов FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ выполняется за время $\Theta(n)$. Поскольку каждая итерация каждого из двух циклов **for** требует $\Theta(1)$ времени, нужно просто подсчитать, сколько всего итераций выполняется. Цикл **for** в строках 3–7 выполняет $mid - low + 1$ итераций, а цикл **for** в строках 10–14 — $high - mid$ итераций, так что общее количество итераций равно

$$
\begin{aligned}
(mid - low + 1) + (high - mid) &= high - low + 1 \\
&= n\ .
\end{aligned}
$$

Имея процедуру FIND-MAX-CROSSING-SUBARRAY с линейным временем работы, можно записать псевдокод алгоритма "разделяй и властвуй", решающего задачу поиска максимального подмассива.

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1  if high == low
2      return (low, high, A[low]) // Базовый случай:
                                  // только один элемент
3  else mid = ⌊(low + high)/2⌋
4      (left-low, left-high, left-sum) =
           FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
           FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
           FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum ≥ right-sum и left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum ≥ left-sum и right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

Начальный вызов FIND-MAXIMUM-SUBARRAY$(A, 1, A.length)$ находит максимальный подмассив массива $A[1\,..\,n]$.

Подобно процедуре FIND-MAX-CROSSING-SUBARRAY, рекурсивная процедура FIND-MAXIMUM-SUBARRAY возвращает кортеж, состоящий из индексов, определяющих максимальный подмассив, и суммы значений этого максимального подмассива. В строке 1 выполняется проверка базового случая — когда подмассив

4 Зак. 3726

состоит только из одного элемента. Подмассив с единственным элементом имеет только один подмассив — себя, так что строка 2 возвращает кортеж с начальным и конечным индексами для единственного элемента вместе с его значением. Строки 3–11 обрабатывают рекурсивный случай. Строка 3 выполняет разделение, вычисляя индекс mid средней точки. Будем называть подмассив $A[low\,..\,mid]$ ***левым подмассивом***, а $A[mid+1\,..\,high]$ — ***правым подмассивом***. Поскольку мы знаем, что подмассив $A[low\,..\,high]$ содержит как минимум два элемента, и левый, и правый подмассивы содержат хотя бы по одному элементу. Строки 4 и 5 "властвуют", рекурсивно находя максимальные подмассивы в левом и правом подмассивах соответственно. Строки 6–11 образуют часть комбинирования. В строке 6 выполняется поиск максимального подмассива, который пересекает среднюю точку. (Вспомним, что, поскольку в строке 6 решается подзадача, не являющаяся экземпляром меньшего размера исходной задачи, мы рассматриваем ее как входящую в часть комбинирования.) В строке 7 выполняется проверка, содержится ли максимальный подмассив в левом подмассиве, и если содержится, то строка 8 возвращает его. В противном случае строка 9 проверяет наличие максимального подмассива в правом подмассиве, а строка 10 возвращает его. Если же ни левый, ни правый подмассивы не содержат подмассива с максимальной суммой, то последний должен пересекать среднюю точку, и он возвращается в строке 11.

Анализ алгоритма "разделяй и властвуй"

Теперь запишем рекуррентное соотношение, которое описывает время работы рекурсивной процедуры FIND-MAXIMUM-SUBARRAY. Как и в ходе анализа сортировки слиянием в разделе 2.3.2, сделаем упрощающее допущение о том, что размер исходной задачи представляет собой степень 2, так что размеры всех подзадач — целые числа. Обозначим время работы FIND-MAXIMUM-SUBARRAY над подмассивом из n элементов как $T(n)$. Для начала строка 1 выполняется за константное время. Базовый случай $n = 1$ прост: строка 2 выполняется за константное время, так что

$$T(1) = \Theta(1) . \tag{4.5}$$

При $n > 1$ осуществляется рекурсивный случай. Строки 1 и 3 выполняются за константное время. Каждая из подзадач, решаемых в строках 4 и 5, работает с подмассивами, состоящими из $n/2$ элементов (наше предположение о том, что размер исходной задачи представляет собой степень 2, гарантирует, что $n/2$ является целым), так что мы тратим время $T(n/2)$ на решение каждой из них. Поскольку мы должны решить две подзадачи (для левого и правого подмассивов), вклад в общее время работы от строк 4 и 5 составляет $2T(n/2)$. Как мы уже видели, вызов FIND-MAX-CROSSING-SUBARRAY в строке 6 требует времени $\Theta(n)$. Строки 7–11 выполняются за время $\Theta(1)$. Итак, для рекурсивного случая мы имеем

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned} \tag{4.6}$$

Объединив уравнения (4.5) и (4.6), мы получаем рекуррентное соотношение для времени работы $T(n)$ процедуры FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) \ , & \text{если } n = 1 \ , \\ 2T(n/2) + \Theta(n) \ , & \text{если } n > 1 \ . \end{cases} \tag{4.7}$$

Это рекуррентное соотношение такое же, как и рекуррентное соотношение (4.1) для сортировки слиянием. Как мы узнаем, познакомившись в разделе 4.5 с основным методом, это рекуррентное соотношение имеет решение $T(n) = \Theta(n \lg n)$. Вы можете также обратиться к дереву рекурсии на рис. 2.5, чтобы понять, почему решение указанного рекуррентного соотношения должно иметь вид $T(n) = \Theta(n \lg n)$.

Таким образом, мы видим, что метод "разделяй и властвуй" дает алгоритм, который оказывается асимптотически быстрее метода грубой силы. Сортировка слиянием, а теперь еще и поиск максимального подмассива демонстрируют нам, насколько мощным может оказаться метод "разделяй и властвуй". Иногда он дает асимптотически самые быстрые алгоритмы для решения задачи, но иногда можно найти еще лучшие алгоритмы. Как показано в упр. 4.1.5, задача поиска максимального подмассива решается за линейное время и не использует метод "разделяй и властвуй".

Упражнения

4.1.1

Что возвращает процедура FIND-MAXIMUM-SUBARRAY, когда все элементы A отрицательны?

4.1.2

Напишите псевдокод для решения задачи поиска максимального подмассива методом грубой силы. Ваша процедура должна выполняться за время $\Theta(n^2)$.

4.1.3

Реализуйте и метод грубой силы, и рекурсивный алгоритм на своем компьютере. Каким оказывается размер задачи точки пересечения n_0, в которой рекурсивный алгоритм превосходит алгоритм грубой силы? Далее измените базовый случай рекуррентного алгоритма, применяя алгоритм грубой силы при размере задачи, не превосходящем n_0. Меняет ли это точку пересечения?

4.1.4

Предположим, что мы меняем определение задачи поиска максимального подмассива, позволяя конечному результату быть пустым массивом и полагая, что сумма значений пустого массива равна нулю. Как бы вы изменили любой алгоритм, не допускающий решения в виде пустого массива, чтобы такой результат в виде пустого подмассива стал возможным?

4.1.5

Воспользуйтесь приведенными далее идеями для разработки нерекурсивного алгоритма поиска максимального подмассива за линейное время. Начните с левого конца массива и двигайтесь вправо, отслеживая найденный к данному моменту максимальный подмассив. Зная максимальный подмассив массива $A[1\,..\,j]$, распространите ответ на поиск максимального подмассива, заканчивающегося индексом $j+1$, воспользовавшись следующим наблюдением: максимальный подмассив массива $A[1\,..\,j+1]$ представляет собой либо максимальный подмассив массива $A[1\,..\,j]$, либо подмассив $A[i\,..\,j+1]$ для некоторого $1 \le i \le j+1$. Определите максимальный подмассив вида $A[i\,..\,j+1]$ за константное время, зная максимальный подмассив, заканчивающийся индексом j.

4.2. Алгоритм Штрассена для умножения матриц

Если вы уже встречались с матрицами, то наверняка знаете, как их перемножать. (В противном случае прочтите раздел Г.1 в приложении Г.) Если $A = (a_{ij})$ и $B = (b_{ij})$ представляют собой квадратные матрицы размером $n \times n$, то элементы c_{ij} их произведения $C = A \cdot B$ определяются для $i, j = 1, 2, \ldots, n$ следующим образом:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \; . \tag{4.8}$$

Нам нужно вычислить n^2 элементов матрицы, каждый из которых представляет собой сумму n значений. Приведенная далее процедура получает в качестве входных данных $n \times n$-матрицы A и B и перемножает их, возвращая их $n \times n$-произведение C. Мы считаем, что каждая матрица имеет атрибут *rows*, указывающий количество строк матрицы.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  Пусть C — новая матрица размером n × n
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Процедура SQUARE-MATRIX-MULTIPLY работает следующим образом. Цикл **for** в строках 3–7 вычисляет элементы каждой строки i, а в пределах данной строки i цикл **for** в строках 4–7 вычисляет каждый из элементов c_{ij} для каждого столбца j. Строка 5 инициализирует c_{ij} значением 0 в начале вычисления суммы

из уравнения (4.8), и каждая итерация цикла **for** в строках 6 и 7 добавляет еще один член из (4.8).

Из-за того, что каждый из циклов в тройной вложенности циклов **for** выполняет ровно n итераций, а каждое выполнение строки 7 занимает константное время, вся процедура SQUARE-MATRIX-MULTIPLY выполняется за время $\Theta(n^3)$.

На первый взгляд, можно решить, что любой алгоритм умножения матриц должен выполняться за время $\Omega(n^3)$, поскольку естественное определение умножения матриц требует именно этого количества умножений. Однако это не так: имеется способ умножения матриц за время $o(n^3)$. В этом разделе вы познакомитесь с замечательным рекурсивным алгоритмом Штрассена для умножения матриц размером $n \times n$. Его время работы составляет $\Theta(n^{\lg 7})$, как будет показано в разделе 4.5. Поскольку $\lg 7$ лежит между 2.80 и 2.81, алгоритм Штрассена выполняется за время $O(n^{2.81})$, что асимптотически лучше простой процедуры SQUARE-MATRIX-MULTIPLY.

Простой алгоритм "разделяй и властвуй"

Для простоты при использовании алгоритма "разделяй и властвуй" для вычисления произведения матриц $C = A \cdot B$ будем считать, что в каждой из этих матриц размером $n \times n$ значение n представляет собой точную степень 2. Мы делаем это предположение потому, что на каждом шаге разделения мы будем разбивать матрицы размером $n \times n$ на четыре матрицы размером $n/2 \times n/2$, и то, что n представляет собой точную степень 2, гарантирует, что пока $n \geq 2$, размерность $n/2$ является целым числом.

Допустим, что мы разделяем каждую из матриц A, B и C на четыре матрицы размером $n/2 \times n/2$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

так что уравнение $C = A \cdot B$ можно переписать как

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} . \tag{4.10}$$

Уравнение (4.10) соответствует четырем уравнениям

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \tag{4.14}$$

В каждом из этих четырех уравнений присутствуют два умножения матриц размером $n/2 \times n/2$ и сложение полученных произведений размером $n/2 \times n/2$. Эти уравнения можно использовать для создания прямолинейного рекурсивного алгоритма "разделяй и властвуй".

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  Пусть C — новая матрица размером n × n
3  if n == 1
4      c11 = a11 · b11
5  else разбиение A, B и C как в (4.9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

Этот псевдокод скрывает одну важную тонкость реализации. Как мы разбиваем матрицы в строке 5? Если мы создаем 12 новых матриц размером $n/2 \times n/2$, то мы платим за это временем $\Theta(n^2)$, затраченным на копирование элементов матриц. В действительности матрицы можно разбивать без копирования. Весь фокус заключается в вычислениях индексов. Мы определяем подматрицу как диапазон индексов строк и диапазон индексов столбцов исходной матрицы. В результате мы представляем подматрицу немного не так, как исходную матрицу, и эта тонкость не указана явно. Преимущество заключается в том, что поскольку мы можем указать подматрицу с помощью индексов, выполнение строки 5 требует только времени $\Theta(1)$ (хотя, как мы увидим в дальнейшем, асимптотическое поведение общего времени работы не зависит от того, будем ли мы копировать матрицы или разбивать их без привлечения дополнительной памяти).

Теперь выведем рекуррентное соотношение, описывающее время работы процедуры SQUARE-MATRIX-MULTIPLY-RECURSIVE. Пусть $T(n)$ — время, необходимое для умножения двух матриц размером $n \times n$ с использованием этой процедуры. В базовом случае, когда $n = 1$, мы выполняем только одно скалярное умножение в строке 4, так что

$$T(1) = \Theta(1) . \tag{4.15}$$

При $n > 1$ осуществляется рекуррентный случай. Как говорилось, разбиение матриц в строке 5 требует при использовании вычисления индексов времени $\Theta(1)$. В строках 6–9 мы восемь раз рекурсивно вызываем процедуру SQUARE-MATRIX-MULTIPLY-RECURSIVE. Поскольку каждый рекурсивный вызов перемножает две матрицы размером $n/2 \times n/2$, его вклад в общее время работы составляет $T(n/2)$, а все восемь вызовов выполняются за время $8T(n/2)$. Следует также учесть четыре сложения матриц в строках 6–9. Каждая из этих матриц содержит $n^2/4$ элементов, так что каждое из сложений матриц требует времени $\Theta(n^2)$. Поскольку количество матричных сложений является константой, общее время

сложения в строках 6–9 равно $\Theta(n^2)$. (Мы вновь прибегаем к вычислению индексов для размещения результатов сложения матриц в корректных позициях матрицы C с накладными расходами $\Theta(1)$ для каждого элемента.) Общее время вычисления в рекуррентном случае, таким образом, равно сумме времени разделения, времен всех рекуррентных вызовов и времени сложения матриц, получающихся после рекуррентных вызовов:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) \ . \end{aligned} \tag{4.16}$$

Заметим, что если мы реализуем разбиение с применением копирования, стоимость которого составляет $\Theta(n^2)$, рекуррентное соотношение не изменится, а следовательно, общее время работы просто увеличится на постоянный множитель.

Объединяя уравнения (4.15) и (4.16), мы получим рекуррентное соотношение для времени работы процедуры SQUARE-MATRIX-MULTIPLY-RECURSIVE:

$$T(n) = \begin{cases} \Theta(1) \ , & \text{если } n = 1 \ , \\ 8T(n/2) + \Theta(n^2) \ , & \text{если } n > 1 \ . \end{cases} \tag{4.17}$$

Как мы увидим из основного метода из раздела 4.5, рекуррентное соотношение (4.17) имеет решение $T(n) = \Theta(n^3)$. Таким образом, этот простой подход "разделяй и властвуй" оказывается ничуть не быстрее прямолинейной процедуры SQUARE-MATRIX-MULTIPLY.

Перед тем как продолжить изучение алгоритма Штрассена, давайте рассмотрим происхождение компонентов уравнения (4.16). Разбиение каждой матрицы $n \times n$ с помощью вычисления индексов требует времени $\Theta(1)$, но у нас разбиваются две матрицы. Хотя можно сказать, что разбиение двух матриц требует времени $\Theta(2)$, константа 2 поглощается Θ-обозначением. Сложение двух матриц со, скажем, k элементами занимает время $\Theta(k)$. Поскольку суммируемые нашим алгоритмом матрицы имеют по $n^2/4$ элементов, можно утверждать, что суммирование каждой пары выполняется за время $\Theta(n^2/4)$. Однако Θ-обозначения вновь поглощают постоянный множитель $1/4$, и мы говорим, что сложение двух матриц размером $n/2 \times n/2$ выполняется за время $\Theta(n^2)$. У нас есть четыре таких сложения, и вновь вместо того, чтобы записать время их выполнения как $\Theta(4n^2)$, мы говорим, что они выполняются за время $\Theta(n^2)$. (Конечно, вы можете заметить, что об этих четырех сложениях можно сказать, что они выполняются за время $\Theta(4n^2/4)$ и что $4n^2/4 = n^2$, но главное в том, что Θ-обозначение поглощает постоянные множители, какими бы они ни были.) Таким образом, в конечном итоге у нас имеются два члена $\Theta(n^2)$, которые мы можем объединить в один.

Однако, когда мы учитываем восемь рекурсивных вызовов, мы не можем просто поглотить постоянный множитель 8. Другими словами, мы должны говорить, что вместе они требуют времени $8T(n/2)$, а не просто $T(n/2)$. Вы можете прочувствовать эту разницу, взглянув на дерево рекурсии на рис. 2.5 для рекуррентного соотношения (2.1) (которое идентично рекуррентному соотношению (4.7)), в котором рекурсивный случай имеет вид $T(n) = 2T(n/2) + \Theta(n)$. Множитель 2

определяет, сколько дочерних узлов имеет каждый узел дерева, что, в свою очередь, определяет количество членов, вносящих вклад в общую сумму на каждом уровне дерева. Если бы мы проигнорировали множитель 8 в уравнении (4.16) или множитель 2 в рекуррентном соотношении (4.1), то дерево рекурсии было бы линейным, а не "кустистым", и каждый уровень добавлял бы в сумму только один член.

Таким образом, следует помнить, что хотя асимптотическая запись поглощает константные мультипликативные множители, рекурсивная запись, такая как $T(n/2)$, этого не делает.

Метод Штрассена

Ключевым моментом метода Штрассена является некоторое уменьшение "кустистости" дерева рекурсии, т.е. вместо восьми рекурсивных умножений матриц $n/2 \times n/2$ он выполняет только семь. Цена устранения одного умножения матриц — несколько дополнительных сложений матриц размером $n/2 \times n/2$, но количество сложений при этом остается константой. Как и ранее, постоянное количество сложений матриц поглощается при записи рекуррентного уравнения для времени работы алгоритма Θ-обозначением.

Метод Штрассена не так очевиден. (Наверное, это наибольшее преуменьшение, сделанное в данной книге.) Он состоит из четырех шагов.

1. Разделить входные матрицы A и B и выходную матрицу C на подматрицы размером $n/2 \times n/2$, как в (4.9). Этот шаг выполняется за время $\Theta(1)$ с помощью вычисления индексов, как и в процедуре SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Создать 10 матриц $S_1, S_2, \ldots, S_{10}$, каждая из которых имеет размер $n/2 \times n/2$ и представляет собой сумму или разность двух матриц, созданных на шаге 1. Все 10 матриц можно создать за время $\Theta(n^2)$.

3. Используя подматрицы, созданные на шаге 1, и 10 матриц, созданных на шаге 2, рекурсивно вычислить семь матричных произведений $P_1, P_2, \ldots, P_7$. Каждая матрица P_i имеет размер $n/2 \times n/2$.

4. Вычислить подматрицы $C_{11}, C_{12}, C_{21}, C_{22}$ результирующей матрицы C путем сложения и вычитания различных комбинаций матриц P_i. Все четыре подматрицы можно вычислить за время $\Theta(n^2)$.

Детально шаги 2–4 будут рассмотрены ниже, но у нас уже имеется достаточно информации для написания рекуррентного соотношения для времени работы метода Штрассена. Будем считать, что, когда размер матрицы n снижается до 1, мы выполняем простое скалярное умножение, как в строке 4 процедуры SQUARE-MATRIX-MULTIPLY-RECURSIVE. При $n > 1$ шаги 1, 2 и 4 выполняются за общее время, равное $\Theta(n^2)$, а шаг 3 требует выполнения семи перемножений матриц размером $n/2 \times n/2$. Следовательно, мы получаем следующее рекуррентное со-

отношение для времени работы $T(n)$ алгоритма Штрассена:

$$T(n) = \begin{cases} \Theta(1) , & \text{если } n = 1 , \\ 7T(n/2) + \Theta(n^2) , & \text{если } n > 1 . \end{cases} \tag{4.18}$$

Мы обменяли одно матричное умножение на фиксированное количество сложений матриц. Когда мы научимся работать с рекуррентными соотношениями и получать их решения, мы увидим, что это ведет к меньшему асимптотическому времени работы. Согласно основному методу из раздела 4.5, рекуррентное соотношение (4.18) имеет решение $T(n) = \Theta(n^{\lg 7})$.

Теперь рассмотрим метод Штрассена более подробно. На шаге 2 мы создаем следующие 10 матриц.

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}$$

Поскольку мы должны 10 раз складывать или вычитать матрицы размером $n/2 \times n/2$, этот шаг выполняется за время $\Theta(n^2)$.

На шаге 3 мы рекурсивно перемножаем $n/2 \times n/2$-матрицы семь раз и вычисляем следующие семь матриц размером $n/2 \times n/2$, каждая из которых представляет собой сумму или разность произведений подматриц A и B.

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 \ = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 \ = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}$$

Заметим, что необходимо выполнять только те умножения, которые приведены в среднем столбце представленных выше уравнений; правый столбец просто показывает, чему равны получающиеся произведения в терминах исходных подматриц, созданных на шаге 1.

Шаг 4 суммирует и вычитает матрицы P_i, созданные на шаге 3, и строит четыре подматрицы размером $n/2 \times n/2$ окончательного произведения C. Мы начинаем с

$$C_{11} = P_5 + P_4 - P_2 + P_6 .$$

Раскрывая правую часть и расписывая каждую подматрицу P_i в отдельной строке, размещая при этом сокращающиеся члены один под другим, мы видим, чему в конечном итоге равна подматрица C_{11}

$$\begin{array}{llllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} & , \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21} &
\end{array}$$

что соответствует уравнению (4.11).

Аналогично мы присваиваем

$$C_{12} = P_1 + P_2 ,$$

так что C_{12} равна

$$\begin{array}{llll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} & , \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} &
\end{array}$$

что соответствует уравнению (4.12).

Установка

$$C_{21} = P_3 + P_4$$

делает C_{21} равной

$$\begin{array}{lll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21}
\end{array}$$

в соответствии с уравнением (4.13).

Наконец мы присваиваем

$$C_{22} = P_5 + P_1 - P_3 - P_7 ,$$

так что C_{22} равна

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12}
\end{array}$$

в соответствии с уравнением (4.14). Всего на шаге 4 мы суммируем или вычитаем $n/2 \times n/2$-матрицы восемь раз, что требует времени $\Theta(n^2)$.

Таким образом, мы видим, что алгоритм Штрассена, состоящий из шагов 1–4, возвращает корректное матричное произведение и что рекуррентное соотношение (4.18) описывает время его работы. Поскольку, как мы узнаем из раздела 4.5, это рекуррентное соотношение имеет решение $T(n) = \Theta(n^{\lg 7})$, метод Штрассена асимптотически быстрее прямолинейной процедуры SQUARE-MATRIX-MULTIPLY. В заключительных замечаниях в конце этой главы рассматриваются некоторые практические аспекты алгоритма Штрассена.

Упражнения

Примечание: хотя в упр. 4.2.3–4.2.5 описаны варианты алгоритма Штрассена, прежде чем приступить к их решению, следует прочесть раздел 4.5.

4.2.1

Воспользуйтесь алгоритмом Штрассена для вычисления произведения матриц

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix} .$$

Покажите, как вы это делаете.

4.2.2

Запишите псевдокод алгоритма Штрассена.

4.2.3

Как модифицировать алгоритм Штрассена для перемножения матриц размером $n \times n$, где n не является точной степенью 2? Покажите, что получающийся в результате алгоритм выполняется за время $\Theta(n^{\lg 7})$.

4.2.4

Чему равно наибольшее k, такое, что если вы можете перемножить 3×3-матрицы с помощью k умножений (не предполагая коммутативности умножения), то вы можете перемножить матрицы размером $n \times n$ за время $o(n^{\lg 7})$? Каким должно быть время работы такого алгоритма?

4.2.5

В. Пан (V. Pan) открыл способ перемножения матриц размером 68×68 с использованием только 132 464 умножений, способ перемножения матриц размером 70×70 с использованием 143 640 умножений и способ перемножения матриц размером 72×72 с использованием 155 424 умножений. Какой из методов дает нам лучшее асимптотическое время работы при его использовании в алгоритме "разделяй и властвуй" для перемножения матриц? Проведите сравнение с алгоритмом Штрассена.

4.2.6

Насколько быстро вы сумеете умножить матрицу размером $kn \times n$ на матрицу размером $n \times kn$, применяя алгоритм Штрассена в качестве подпрограммы? Ответьте на тот же вопрос для ситуации, когда мы меняем входные матрицы местами.

4.2.7

Покажите, как перемножить комплексные числа $a + bi$ и $c + di$, используя только три умножения действительных чисел. Алгоритм должен получать a, b, c и d в качестве входных данных и возвращать действительную $(ac - bd)$ и мнимую $(ad + bc)$ части произведения по отдельности.

4.3. Метод подстановки решения рекуррентных соотношений

Теперь, когда вы познакомились с описанием времени работы алгоритмов "разделяй и властвуй" рекуррентными соотношениями, рассмотрим способы решения таких рекуррентных соотношений. В этом разделе начнем рассмотрение с метода подстановки.

Метод подстановки для решения рекуррентных соотношений состоит из двух шагов:

1. делается предположение о виде решения;
2. с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Название "метод подстановки" связано с тем, что мы подставляем предполагаемое решение вместо функции при применении гипотезы индукции для меньших значений. Это мощный метод, но для его применения нужно суметь сделать предположение о виде решения.

Метод подстановки можно применять для определения либо верхней, либо нижней границы рекуррентного соотношения. В качестве примера определим верхнюю границу рекуррентного соотношения

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ , \tag{4.19}$$

подобного соотношениям (4.3) и (4.4). Мы предполагаем, что решение имеет вид $T(n) = O(n \lg n)$. Наш метод заключается в доказательстве того, что при подходящем выборе константы $c > 0$ выполняется неравенство $T(n) \leq cn \lg n$. Начнем с того, что предположим справедливость этого неравенства для всех положительных $m < n$, в частности для $m = \lfloor n/2 \rfloor$, что дает $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.

Подстановка в рекуррентное соотношение приводит к

$$\begin{aligned}T(n) &\leq 2(c\lfloor n/2\rfloor \lg(\lfloor n/2\rfloor)) + n \\ &\leq cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\leq cn\lg n\ ,\end{aligned}$$

где последний шаг выполняется при $c \geq 1$.

Теперь, согласно методу математической индукции, необходимо доказать, что наше решение справедливо для граничных условий. Обычно для этого достаточно показать, что граничные условия являются подходящей базой для доказательства по индукции. В рекуррентном соотношении (4.19) необходимо доказать, что константу c можно выбрать достаточно большой для того, чтобы соотношение $T(n) \leq cn\lg n$ было справедливо и для граничных условий. Такое требование иногда приводит к проблемам. Предположим, например, что $T(1) = 1$ — единственное граничное условие рассматриваемого рекуррентного соотношения. Далее, для $n = 1$ соотношение $T(n) \leq cn\lg n$ дает нам $T(1) \leq c \cdot 1 \cdot \lg 1 = 0$, что противоречит условию $T(1) = 1$. Следовательно, данный базис индукции нашего доказательства не выполняется.

Эту сложность, возникающую при доказательстве предположения индукции для указанного граничного условия, легко обойти. Например, в рекуррентном соотношении (4.19) можно воспользоваться преимуществами асимптотических обозначений, требующих доказать неравенство $T(n) \leq cn\lg n$ для $n \geq n_0$, где n_0 — *выбранная нами* константа. Идея по устранению возникшей проблемы заключается в том, чтобы в доказательстве по методу математической индукции не учитывать граничное условие $T(1) = 1$. Обратите внимание, что при $n > 3$ рассматриваемое рекуррентное соотношение явным образом от $T(1)$ не зависит. Таким образом, выбрав $n_0 = 2$, в качестве базы индукции можно рассматривать не $T(1)$, а $T(2)$ и $T(3)$. Заметим, что здесь делается различие между базой рекуррентного соотношения ($n = 1$) и базой индукции ($n = 2$ и $n = 3$). Из рекуррентного соотношения следует, что $T(2) = 4$, а $T(3) = 5$. Теперь доказательство по методу математической индукции соотношения $T(n) \leq cn\lg n$ для некоторой константы $c \geq 1$ можно завершить, выбрав ее достаточно большой для того, чтобы были справедливы неравенства $T(2) \leq c2\lg 2$ и $T(3) \leq c3\lg 3$. Оказывается, что для этого достаточно выбрать $c \geq 2$. В большинстве рекуррентных соотношений, которые нам предстоит рассмотреть, легко расширить граничные условия таким образом, чтобы гипотеза индукции оказалась верна для малых n.

Как угадать решение

К сожалению, не существует общего способа, позволяющего угадать правильное решение рекуррентного соотношения. Для этого требуются опыт, удача и творческое мышление. К счастью, существуют определенные эвристические приемы, которые могут помочь сделать правильную догадку. Кроме того, для

получения предполагаемого вида решения можно воспользоваться деревьями рекурсии, с которыми мы познакомимся в разделе 4.4.

Если рекуррентное соотношение подобно тому, которое мы уже рассматривали, то разумно предположить, что решения этих соотношений будут похожими. Например, рассмотрим рекуррентное соотношение

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \ ,$$

которое выглядит более сложным, поскольку в аргументе функции T в его правой части добавлено слагаемое "17". Однако интуитивно понятно, что это дополнительное слагаемое не может сильно повлиять на асимптотическое поведение решения. При достаточно больших n разность между $\lfloor n/2 \rfloor$ и $\lfloor n/2 \rfloor + 17$ невелика: оба эти числа приблизительно равны половине числа n. Следовательно, можно предположить, что $T(n) = O(n \lg n)$; проверить корректность этого предположения можно методом подстановки (см. упр. 4.3.6).

Другой способ найти решение — выполнить грубую оценку его верхней и нижней границ, а затем свести неопределенность до минимальной. Например, в рекуррентном соотношении (4.19) в качестве начальной нижней границы можно было бы выбрать $T(n) = \Omega(n)$, поскольку в нем содержится слагаемое n; можно также доказать, что грубой верхней границей является $T(n) = O(n^2)$. Далее верхняя граница постепенно понижается, а нижняя — повышается до тех пор, пока не будет получено правильное асимптотическое поведение решения $T(n) = \Theta(n \lg n)$.

Тонкие нюансы

Иногда сделать правильное предположение об асимптотическом поведении решения рекуррентного соотношения можно, но при этом возникают трудности, связанные с выполнением доказательства по методу математической индукции. Обычно проблема заключается в том, что выбрано недостаточно сильное предположение индукции, которое не позволяет доказать точную границу. Натолкнувшись на такое препятствие, пересмотрите предположение индукции, избавившись от членов низшего порядка. При этом часто удается провести строгое математическое доказательство.

Рассмотрим рекуррентное соотношение

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \ .$$

Можно предположить, что его решением является $T(n) = O(n)$, и попытаться показать, что $T(n) \le cn$ для подходящего выбора константы c. Подставив предполагаемое решение в рекуррентное соотношение, получим выражение

$$\begin{aligned} T(n) &\le c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 \ , \end{aligned}$$

из которого не следует $T(n) \le cn$ ни для какого выбора c. Можно попытаться сделать другие предположения, например о том, что решение — $T(n) = O(n^2)$.

Хотя это предположение и можно доказать, наше первоначальное предположение вполне корректно. Однако чтобы это показать, необходимо выбрать более сильную гипотезу индукции.

Интуитивно понятно, что наша догадка была почти правильной: мы ошиблись всего лишь на константу, равную 1, т.е. на величину низшего порядка. Тем не менее математическая индукция не работает, если в предположении индукции допущена даже такая, казалось бы, незначительная ошибка. Эту трудность можно преодолеть, если *вычесть* из первоначального предположения член низшего порядка. Таким образом, теперь гипотеза индукции имеет вид $T(n) \leq cn - d$, где $b \geq 0$ – константа. Теперь мы имеем

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \ , \end{aligned}$$

которое справедливо при $d \geq 1$. Как и раньше, чтобы выполнялись граничные условия, константу c необходимо выбрать достаточно большой.

Идея вычитания члена более низкого порядка может показаться противоречащей интуитивным представлениям. Ведь если предположение не сработало, его следует ослабить, не так ли? Не обязательно! При доказательстве верхней границы по индукции в действительности может оказаться сложнее доказать более слабую верхнюю границу, поскольку при ее доказательстве мы вынуждены использовать в доказательстве ее же. В наших примерах, в которых рекуррентное соотношение имеет более одного рекурсивного члена, мы вычитаем член более низкого порядка из предложенной границы по одному разу для каждого рекурсивного члена. В приведенном выше примере константа d вычитается дважды, один раз для члена $T(\lfloor n/2 \rfloor)$ и второй – для члена $T(\lceil n/2 \rceil)$. В результате получается неравенство $T(n) \leq cn - 2d + 1$, и можно легко найти значения d, которые делают $cn - 2d + 1$ меньшим или равным $cn - d$.

Остерегайтесь ошибок

Используя асимптотические обозначения, легко допустить ошибку. Например, для рекуррентного соотношения (4.19) легко "доказать", что $T(n) = O(n)$, предположив, что $T(n) \leq cn$, а затем рассуждая следующим образом:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \ , \qquad \Longleftarrow \textit{Ошибка!!} \end{aligned}$$

поскольку c представляет собой константу. Ошибка заключается в том, что не была доказана гипотеза индукции в *точном виде*, т.е. что $T(n) \leq cn$. Таким образом, мы неявно доказали, что $T(n) \leq cn$, в то время как мы хотели показать, что $T(n) = O(n)$.

Замена переменных

Иногда с помощью небольших алгебраических преобразований удается добиться того, что неизвестное рекуррентное соотношение становится похожим на то, с которым мы уже знакомы. Например, рассмотрим рекуррентное соотношение

$$T(n) = 2T\left(\lfloor\sqrt{n}\rfloor\right) + \lg n\ ,$$

которое выглядит довольно сложным. Однако его можно упростить, выполнив замену переменных. Для удобства мы не станем беспокоиться об округлении до целых таких значений, как $\sqrt{n}$. Переименование $m = \lg n$ дает

$$T(2^m) = 2T(2^{m/2}) + m\ .$$

Теперь можно переименовать $S(m) = T(2^m)$, чтобы получить новое рекуррентное соотношение

$$S(m) = 2S(m/2) + m\ ,$$

которое очень похоже на рекуррентное соотношение (4.19). Это новое рекуррентное соотношение имеет то же самое решение: $S(m) = O(m \lg m)$. Выполнив обратную замену $S(m)$ на $T(n)$, мы получаем

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg\lg n)\ .$$

Упражнения

4.3.1

Покажите, что решением $T(n) = T(n-1) + n$ является $O(n^2)$.

4.3.2

Покажите, что решением $T(n) = T(\lceil n/2 \rceil) + 1$ является $O(\lg n)$.

4.3.3

Мы видели, что решением $T(n) = 2T(\lfloor n/2 \rfloor) + n$ является $O(n \lg n)$. Покажите, что решение этого рекуррентного соотношения также представляет собой $\Omega(n \lg n)$. Сделайте вывод, что решением рассматриваемого рекуррентного соотношения является $\Theta(n \lg n)$.

4.3.4

Покажите, что преодолеть трудность, связанную с граничным условием $T(1) = 1$ в рекуррентном соотношении (4.19) можно путем выбора другого предположения индукции, не меняя при этом граничных условий.

4.3.5

Покажите, что $\Theta(n \lg n)$ является решением “точного” рекуррентного соотношения (4.3) для сортировки слиянием.

4.3.6

Покажите, что решением рекуррентности $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ является $O(n \lg n)$.

4.3.7

Используя основной метод из раздела 4.5, можно показать, что решением рекуррентного соотношения $T(n) = 4T(n/3)+n$ является $T(n) = \Theta(n^{\log_3 4})$. Покажите, что не получается выполнить доказательство методом подстановок с гипотезой индукции $T(n) \le cn^{\log_3 4}$. Затем покажите, как вычитание члена меньшего порядка делает доказательство возможным.

4.3.8

Используя основной метод из раздела 4.5, можно показать, что решением рекуррентного соотношения $T(n) = 4T(n/2) + n$ является $T(n) = \Theta(n^2)$. Покажите, что не получается выполнить доказательство методом подстановок с гипотезой индукции $T(n) \le cn^2$. Затем покажите, как вычитание члена меньшего порядка делает доказательство возможным.

4.3.9

Решите рекуррентное соотношение $T(n) = 3T(\sqrt{n}) + \log n$ путем замены переменных. Ваше решение должно быть асимптотически точным. При решении не беспокойтесь о том, чтобы все значения являлись целыми числами.

4.4. Метод деревьев рекурсии

Метод подстановок способен обеспечить краткий путь к доказательству того факта, что предполагаемое решение рекуррентного соотношения является правильным, однако сделать хорошую догадку зачастую довольно трудно. Построение дерева рекурсии, подобного тому, с которым мы имели дело в разделе 2.3.2 в ходе анализа рекуррентного соотношения, описывающего время сортировки слиянием, — прямой путь к хорошей догадке. В ***дереве рекурсии*** каждый узел представляет стоимость выполнения отдельно взятой подзадачи, которая решается при одном из многочисленных рекурсивных вызовов функций. Далее стоимости отдельных этапов суммируются в пределах каждого уровня, а затем — по всем уровням дерева, в результате чего получаем полную стоимость всех уровней рекурсии.

Деревья рекурсии лучше всего подходят для того, чтобы помочь сделать догадку о виде решения, которая затем проверяется методом подстановок. При этом в догадке часто допускается наличие небольших неточностей, поскольку впоследствии она все равно проверяется. Если же построение дерева рекурсии и суммирование времени работы по всем его составляющим производится достаточно тщательно, то само дерево рекурсии может стать средством доказательства корректности решения. В данном разделе деревья рекурсии применяются для полу-

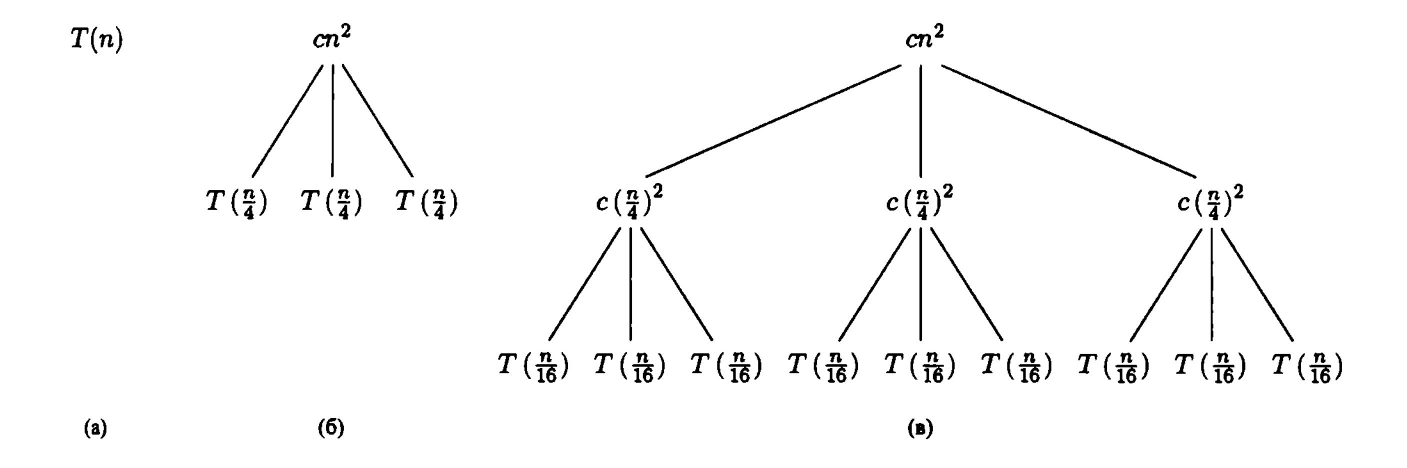
чения предположений о виде решения, а в разделе 4.6 — непосредственно для доказательства теоремы, на которой базируется основной метод.

Например, посмотрим, как с помощью дерева рекурсии можно догадаться о виде решения рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Начнем с поиска верхней границы решения. Как известно, при решении рекуррентных соотношений тот факт, что от аргументов функций берется целая часть, при решении рекуррентных соотношений обычно является несущественным (это пример отклонений, которые можно допустить), поэтому построим дерево рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$, записанного с использованием константы $c > 0$.

На рис. 4.5 проиллюстрирован процесс построения дерева рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$. Для удобства предположим, что n — степень четверки (еще один пример допустимых отклонений), так что размеры всех подзадач — целые. В части (а) показана функция $T(n)$, которая затем, в части (б), раскрывается в эквивалентное дерево рекурсии, представляющее анализируемое рекуррентное соотношение. Член cn^2 в корне дерева представляет время верхнего уровня рекурсии, а три поддерева, берущих начало из корня, — времена выполнения подзадач размером $n/4$. В части (в) добавлен еще один шаг раскрытия, т.е. выполнено представление в виде поддерева каждого узла с временем $T(n/4)$ из части (б). Время выполнения, соответствующее каждому из трех дочерних поддеревьев, равно $c(n/4)^2$. Далее каждый лист дерева преобразуется в поддерево аналогичным образом в соответствии с рекуррентным соотношением.

Поскольку по мере удаления от корня дерева размер подзадач уменьшается на каждом уровне в четыре раза, в конце концов мы должны дойти до граничных условий. Сколько же уровней дерева нужно построить, чтобы их достичь? Размер вспомогательной задачи, соответствующей уровню, который находится на i-м уровне глубины, равен $n/4^i$. Таким образом, размер подзадачи достигает $n = 1$, когда $n/4^i = 1$ или, что то же самое, когда $i = \log_4 n$. Таким образом, всего в дереве $\log_4 n + 1$ уровней (с глубинами $0, 1, 2, \ldots, \log_4 n$).

Затем мы определяем стоимость каждого уровня дерева. На каждом уровне в три раза больше узлов, чем на предыдущем, поэтому количество узлов на i-м уровне равно 3^i. Поскольку размеры вспомогательных подзадач при спуске на один уровень уменьшаются в четыре раза, время выполнения каждого узла на i-м уровне (для $i = 0, 1, 2, \ldots, \log_4 n - 1$) равно $3^i c(n/4^i)^2 = (3/16)^i cn^2$. На нижнем уровне на глубине $\log_4 n$ имеется $3^{\log_4 n} = n^{\log_4 3}$ узлов, каждый из которых дает в общее время работы вклад, равный $T(1)$. Поэтому время работы этого уровня равно величине $n^{\log_4 3}T(1)$, что представляет собой $\Theta(n^{\log_4 3})$, так как мы считаем, что $T(1)$ является константой.



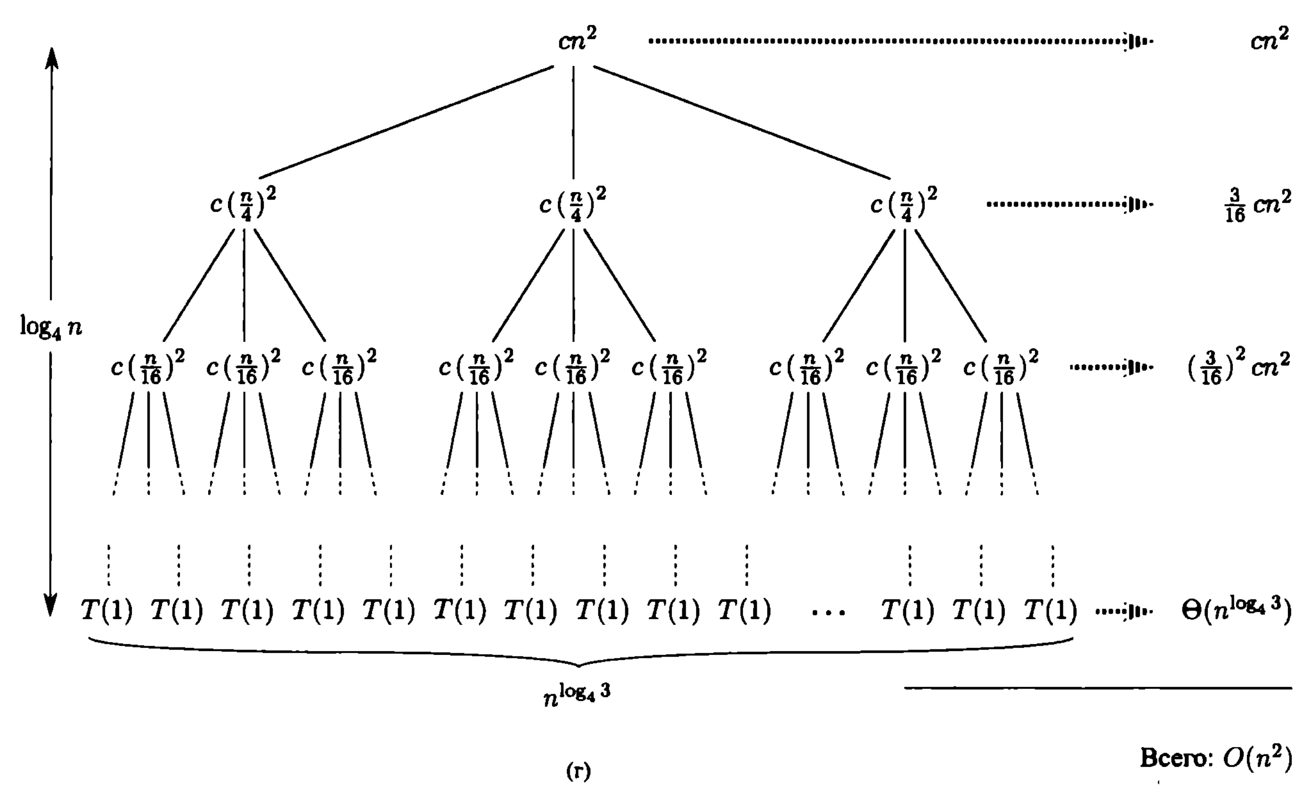


Рис. 4.5. Построение дерева рекурсии для $T(n) = 3T(n/4) + cn^2$. В части **(а)** показано значение $T(n)$, которое постепенно раскрывается в частях **(б)–(г)**, образуя дерево рекурсии. Полностью раскрытое дерево в части **(г)** имеет высоту $\log_4 n$ (в нем $\log_4 n + 1$ уровней).

Теперь просуммируем стоимости всех уровней, чтобы найти стоимость всего дерева:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(согласно (A.5))} .
\end{aligned}$$

Эта формула выглядит несколько запутанной, но только до тех пор, пока мы не догадаемся воспользоваться той небольшой свободой, которая допускается при асимптотических оценках, и не используем в качестве верхней границы одного из слагаемых бесконечно убывающую геометрическую прогрессию. Возвращаясь на один шаг назад и применяя формулу (А.6), получаем

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)}\, cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13}\, cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2)\ . \end{aligned}$$

Таким образом, для исходного рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ получен предполагаемый вид решения $T(n) = O(n^2)$. В рассматриваемом примере коэффициенты при cn^2 образуют убывающую геометрическую прогрессию, а их сумма, согласно уравнению (А.6), ограничена сверху константой $16/13$. Поскольку вклад корня дерева в полное время работы равен cn^2, время работы корня представляет собой некоторую постоянную часть от общего времени работы всего дерева в целом. Другими словами, полное время работы всего дерева в основном определяется временем работы его корня.

В действительности, если $O(n^2)$ в самом деле представляет собой верхнюю границу для рекуррентного соотношения (в чем мы вскоре убедимся), эта граница должна быть асимптотически точной оценкой. Почему? Потому что первый рекурсивный вызов дает вклад в общее время работы алгоритма, который выражается как $\Theta(n^2)$, поэтому нижняя граница решения рекуррентного соотношения представляет собой $\Omega(n^2)$.

Теперь, чтобы убедиться в том, что наше предположение верно, т.е. что $T(n) = O(n^2)$ является верхней границей для рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$, можно воспользоваться методом подстановок. Мы хотим показать, что $T(n) \le dn^2$ для некоторой константы $d > 0$. Используя ту же константу $c > 0$, что и ранее, мы получаем

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}\, dn^2 + cn^2 \\ &\le dn^2\ , \end{aligned}$$

где последний шаг выполняется при $d \ge (16/13)c$.

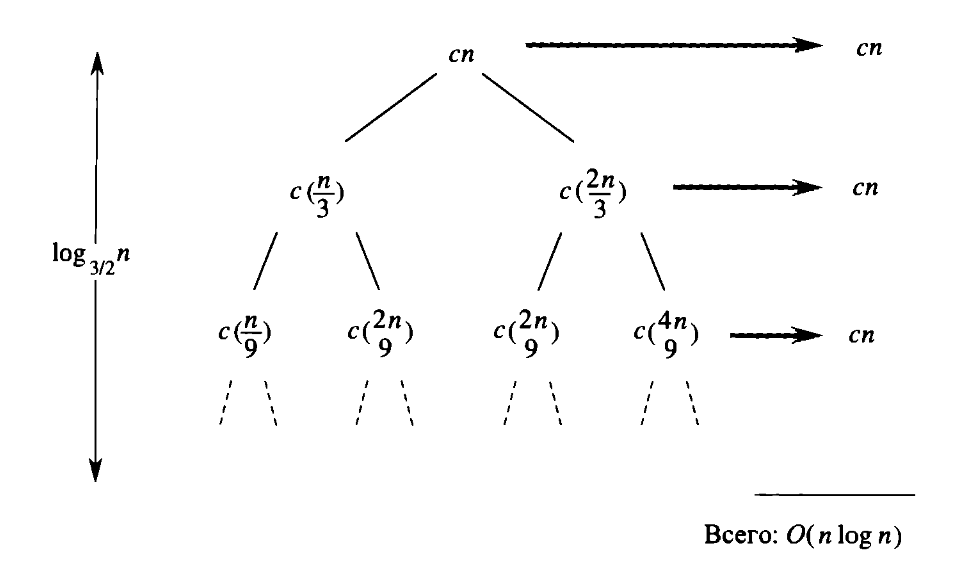


Рис. 4.6. Дерево рекурсии для рекуррентного соотношения $T(n) = T(n/3) + T(2n/3) + cn$.

Давайте теперь рассмотрим более сложный пример. На рис. 4.6 представлено дерево рекурсии для рекуррентного соотношения

$$T(n) = T(n/3) + T(2n/3) + O(n) .$$

(Здесь также для простоты опущены функции "пол" и "потолок".) Пусть c, как и ранее, представляет постоянный множитель в члене $O(n)$. При суммировании величин по всем узлам дерева, принадлежащим определенному уровню, для каждого уровня мы получаем величину cn. Самый длинный путь от корня дерева до его листа имеет вид $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Поскольку $(2/3)^k n = 1$ при $k = \log_{3/2} n$, высота дерева равна $\log_{3/2} n$.

Интуитивно мы ожидаем, что решение рекуррентного соотношения не превосходит количества уровней, умноженного на стоимость каждого уровня, или $O(cn \log_{3/2} n) = O(n \lg n)$. На рис. 4.6 показаны только верхние уровни дерева рекурсии, и не каждый уровень дерева дает вклад cn. Рассмотрим стоимость листьев. Если это дерево рекурсии представляет собой полное бинарное дерево высотой $\log_{3/2} n$, в нем должно иметься $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ листьев. Поскольку стоимость каждого листа является константой, общая стоимость листьев должна составлять $\Theta(n^{\log_{3/2} 2})$, что является $\omega(n \lg n)$, поскольку $\log_{3/2} 2$ представляет собой константу, строго большую, чем 1. Однако это дерево рекурсии не является полным бинарным деревом, а потому имеет менее $n^{\log_{3/2} 2}$ листьев. Более того, по мере удаления от корня отсутствует все большее количество внутренних узлов. Следовательно, не для всех уровней время их работы выражается как cn; более низкие уровни дают меньший вклад. Можно было бы попытаться аккуратно учесть вклады всех элементов дерева, однако вспомним, что мы всего лишь угадываем вид решения, чтобы затем воспользоваться методом подстановок. Давайте отклонимся от точного решения и допустим, что асимптотическая верхняя граница решения представляет собой $O(n \lg n)$.

Действительно, можно использовать метод подстановок, чтобы проверить, что $O(n \lg n)$ является верхней границей решения рекуррентного соотношения. Покажем, что $T(n) \le dn \lg n$, где d — подходящая положительная константа. Мы

имеем

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\quad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn\lg n
\end{aligned}
$$

при $d \geq c/(\lg 3 - (2/3))$. Таким образом, нам не нужно выполнять более точный учет стоимости в дереве рекурсии.

Упражнения

4.4.1

Воспользуйтесь деревом рекурсии для определения точной асимптотической верхней границы рекуррентного соотношения $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Для проверки своего ответа используйте метод подстановок.

4.4.2

Воспользуйтесь деревом рекурсии для определения точной асимптотической верхней границы рекуррентного соотношения $T(n) = T(n/2) + n^2$. Для проверки своего ответа используйте метод подстановок.

4.4.3

Воспользуйтесь деревом рекурсии для определения точной асимптотической верхней границы рекуррентного соотношения $T(n) = 4T(n/2 + 2) + n$. Для проверки своего ответа используйте метод подстановок.

4.4.4

Воспользуйтесь деревом рекурсии для определения точной асимптотической верхней границы рекуррентного соотношения $T(n) = 2T(n-1) + 1$. Для проверки своего ответа используйте метод подстановок.

4.4.5

Воспользуйтесь деревом рекурсии для определения точной асимптотической верхней границы рекуррентного соотношения $T(n) = T(n-1) + T(n/2) + n$. Для проверки своего ответа используйте метод подстановок.

4.4.6

Обратившись к соответствующему дереву рекурсии, докажите, что решением рекуррентного соотношения $T(n) = T(n/3) + T(2n/3) + cn$, где c представляет собой константу, является $\Omega(n \lg n)$.

4.4.7

Постройте дерево рекурсии для рекуррентного соотношения $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, где c — константа, и найдите точную асимптотическую границу его решения. Проверьте ее с помощью метода подстановок.

4.4.8

Найдите с помощью дерева рекурсии точную асимптотическую оценку решения рекуррентного соотношения $T(n) = T(n-a) + T(a) + cn$, где $a \geq 1$ и $c > 0$ являются константами.

4.4.9

Найдите с помощью дерева рекурсии точную асимптотическую оценку решения рекуррентного соотношения $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, где α — константа из диапазона $0 < \alpha < 1$; константой является и $c > 0$.

4.5. Основной метод

Основной метод является своего рода "кулинарной книгой", по которой строятся решения рекуррентных соотношений вида

$$T(n) = aT(n/b) + f(n) , \tag{4.20}$$

где $a \geq 1$ и $b > 1$ — константы, а $f(n)$ — асимптотически положительная функция. Чтобы научиться пользоваться этим методом, необходимо запомнить три случая, что значительно облегчает решение многих рекуррентных соотношений, а часто это можно сделать даже в уме.

Рекуррентное соотношение (4.20) описывает время работы алгоритма, в котором задача размером n разбивается на a вспомогательных задач размером n/b каждая, где a и b — положительные константы. Полученные в результате разбиения a подзадач решаются рекурсивно, причем время решения каждой из них равно $T(n/b)$. Время, требуемое для разбиения задачи и объединения результатов, полученных при решении вспомогательных задач, описывается функцией $f(n)$. Например, в рекуррентном соотношении, возникающем в ходе анализа алгоритма Штрассена, $a = 7$, $b = 2$, а $f(n) = \Theta(n^2)$.

Строго говоря, при определении приведенного выше рекуррентного соотношения допущена неточность, поскольку число n/b может не быть целым. Однако замена каждого из a слагаемых $T(n/b)$ выражением $T(\lfloor n/b \rfloor)$ или $T(\lceil n/b \rceil)$ не влияет на асимптотическое поведение решения (это будет доказано в следующем разделе). Поэтому обычно при составлении рекуррентных соотношений подобного вида, полученных методом "разделяй и властвуй", мы будем игнорировать функции "пол" и "потолок".

Основная теорема

Основной метод зависит от следующей теоремы.

Теорема 4.1 (основная теорема)

Пусть $a \geq 1$ и $b > 1$ — константы, $f(n)$ — функция, а $T(n)$ определена на множестве неотрицательных целых чисел с помощью рекуррентного соотношения

$$T(n) = aT(n/b) + f(n) ,$$

где n/b интерпретируется либо как $\lfloor n/b \rfloor$, либо как $\lceil n/b \rceil$. Тогда $T(n)$ имеет следующие асимптотические границы.

1. Если $f(n) = O(n^{\log_b a - \epsilon})$ для некоторой константы $\epsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Если $f(n) = \Omega(n^{\log_b a + \epsilon})$ для некоторой константы $\epsilon > 0$ и если $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$ и всех достаточно больших n, то $T(n) = \Theta(f(n))$. ■

Прежде чем применить основную теорему к некоторым конкретным примерам, потратим немного времени на понимание ее сути. В каждом из трех случаев функция $f(n)$ сравнивается с функцией $n^{\log_b a}$. Интуитивно понятно, что бóльшая из этих двух функций определяет решение рекуррентного соотношения. Если, как в случае 1, функция $n^{\log_b a}$ больше, то решение имеет вид $T(n) = \Theta(n^{\log_b a})$. Если, как в случае 3, большей является функция $f(n)$, то решение представляет собой $T(n) = \Theta(f(n))$. Если же, как в случае 2, две эти функции сравнимы, мы выполняем умножение на логарифмический множитель, и решением является $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Помимо интуитивных представлений, нужно знать и некоторые технические детали. В первом случае функция $f(n)$ должна быть не просто меньше, чем $n^{\log_b a}$, а *полиномиально* меньше (т.е. $f(n)$ должна быть асимптотически меньше $n^{\log_b a}$ на множитель n^ϵ для некоторой константы $\epsilon > 0$). В третьем случае функция $f(n)$ должна быть не просто больше $n^{\log_b a}$, но полиномиально больше, и к тому же удовлетворять условию "регулярности" $af(n/b) \leq cf(n)$. Это условие удовлетворяется большинством полиномиально ограниченных функций, с которыми нам предстоит встречаться.

Важно понимать, что этими тремя случаями не исчерпываются все возможности поведения функции $f(n)$. Между случаями 1 и 2 есть промежуток, в котором функция $f(n)$ меньше функции $n^{\log_b a}$, но не полиномиально меньше. Аналогичный промежуток имеется между случаями 2 и 3, когда функция $f(n)$ больше функции $n^{\log_b a}$, но не полиномиально больше. Если функция $f(n)$ попадает в один из этих промежутков или если для нее не выполняется условие регулярности из случая 3, для решения таких рекуррентных соотношений основной метод неприменим.

Использование основного метода

При использовании основного метода мы просто определяем, какой из случаев основной теоремы (если таковой имеет место) применим в данной ситуации, и записываем ответ.

В качестве первого примера рассмотрим

$$T(n) = 9T(n/3) + n \ .$$

В этом рекуррентном соотношении мы имеем $a = 9$, $b = 3$, $f(n) = n$, так что $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Поскольку $f(n) = O(n^{\log_3 9 - \epsilon})$, где $\epsilon = 1$, мы можем применить случай 1 основной теоремы и сделать вывод, что решение имеет вид $T(n) = \Theta(n^2)$.

Рассмотрим теперь рекуррентное соотношение

$$T(n) = T(2n/3) + 1,$$

в котором $a = 1$, $b = 3/2$, $f(n) = 1$ и $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Здесь применим случай 2, поскольку $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, и, таким образом, решение рекуррентного соотношения представляет собой $T(n) = \Theta(\lg n)$.

В случае рекуррентного соотношения

$$T(n) = 3T(n/4) + n \lg n$$

мы имеем $a = 3$, $b = 4$, $f(n) = n \lg n$ и $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Поскольку $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, где $\epsilon \approx 0.2$, можно применить случай 3, если мы сможем показать, что для функции $f(n)$ выполняется условие регулярности. При достаточно больших n мы получаем, что $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ для $c = 3/4$. Следовательно, в соответствии со случаем 3 решением рекуррентного соотношения является $T(n) = \Theta(n \lg n)$.

Основной метод неприменим к рекуррентному соотношению

$$T(n) = 2T(n/2) + n \lg n$$

несмотря на то, что оно выглядит как имеющее корректный вид: $a = 2$, $b = 2$, $f(n) = n \lg n$ и $n^{\log_b a} = n$. Вы можете ошибочно решить, что к нему применим случай 3, поскольку $f(n) = n \lg n$ асимптотически больше, чем $n^{\log_b a} = n$. Проблема в том, что данная функция больше не *полиномиально*. Отношение $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ асимптотически меньше n^ϵ для любой положительной константы ϵ. Таким образом, рассматриваемое рекуррентное соотношение попадает в "зазор" между случаями 2 и 3. (См. решение этого рекуррентного соотношения в упр. 4.6.2.)

Применим основной метод к решению рекуррентных соотношений, с которыми мы встречались в разделах 4.1 и 4.2. Рекуррентное соотношение (4.7),

$$T(n) = 2T(n/2) + \Theta(n) \ ,$$

характеризует время работы алгоритмов "разделяй и властвуй" для задачи поиска максимального подмассива и сортировки слиянием. (Как обычно для практических целей, мы опускаем базовый случай рекуррентного соотношения.) Здесь мы имеем $a = 2$, $b = 2$, $f(n) = \Theta(n)$ и, таким образом, получаем, что $n^{\log_b a} =$

$n^{\log_2 2} = n$. Применим случай 2, поскольку $f(n) = \Theta(n)$, так что решением рекуррентного соотношения является $T(n) = \Theta(n \lg n)$.

Рекуррентное соотношение (4.17),

$$T(n) = 8T(n/2) + \Theta(n^2) ,$$

описывает время работы первого алгоритма "разделяй и властвуй" для матричного умножения. Мы имеем $a = 8$, $b = 2$ и $f(n) = \Theta(n^2)$, так что $n^{\log_b a} = n^{\log_2 8} = n^3$. Поскольку n^3 полиномиально больше $f(n)$ (т.е. $f(n) = O(n^{3-\epsilon})$ для $\epsilon = 1$), применим случай 1, и $T(n) = \Theta(n^3)$.

Наконец рассмотрим рекуррентное соотношение (4.18),

$$T(n) = 7T(n/2) + \Theta(n^2) ,$$

которое описывает время работы алгоритма Штрассена. В этом случае $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, так что $n^{\log_b a} = n^{\log_2 7}$. Переписав $\log_2 7$ как $\lg 7$ и вспомнив, что $2.80 < \lg 7 < 2.81$, мы видим, что $f(n) = O(n^{\lg 7-\epsilon})$ для $\epsilon = 0.8$. Вновь можно применить случай 1 и получить решение $T(n) = \Theta(n^{\lg 7})$.

Упражнения

4.5.1

С помощью основной теоремы найдите точные асимптотические границы следующих рекуррентных соотношений.

а. $T(n) = 2T(n/4) + 1$.

б. $T(n) = 2T(n/4) + \sqrt{n}$.

в. $T(n) = 2T(n/4) + n$.

г. $T(n) = 2T(n/4) + n^2$.

4.5.2

Профессор хочет разработать алгоритм матричного умножения, асимптотически более быстрый, чем алгоритм Штрассена. Его алгоритм будет использовать метод "разделяй и властвуй", разбивая каждую матрицу на части размером $n/4 \times n/4$, причем шаги разделения и комбинирования выполняются за время $\Theta(n^2)$. Профессору требуется определить, сколько подзадач должен создавать его алгоритм, чтобы опередить алгоритм Штрассена. Если алгоритм профессора создает a подзадач, то рекуррентное соотношение для времени работы $T(n)$ принимает вид $T(n) = aT(n/4) + \Theta(n^2)$. Каково наибольшее целочисленное значение a, для которого алгоритм профессора оказывается асимптотически быстрее алгоритма Штрассена?

4.5.3

Покажите с помощью основного метода, что $T(n) = \Theta(\lg n)$ является решением рекуррентного соотношения $T(n) = T(n/2) + \Theta(1)$, возникающего в ходе анализа алгоритма бинарного поиска. (Алгоритм бинарного поиска описан в упр. 2.3.5.)

4.5.4

Можно ли применить основной метод к рекуррентному соотношению $T(n) = 4T(n/2) + n^2 \lg n$? Обоснуйте свой ответ. Найдите асимптотическую верхнюю границу решения этого рекуррентного соотношения.

4.5.5 ★

Рассмотрим условие регулярности $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$, являющееся частью случая 3 основной теоремы. Приведите примеры констант $a \geq 1$ и $b > 1$ и функции $f(n)$, которые удовлетворяют всем условиям случая 3 основной теоремы, кроме условия регулярности.

★ 4.6. Доказательство основной теоремы

В этом разделе содержится доказательство основной теоремы (теоремы 4.1). Для применения самой теоремы понимать доказательство необязательно.

Доказательство состоит из двух частей. В первой части анализируется "основное" рекуррентное соотношение (4.20) с учетом упрощающего предположения, согласно которому $T(n)$ определена только для точных степеней числа $b > 1$, т.е. для $n = 1, b, b^2, \ldots$. В этой части представлены все основные идеи, необходимые для доказательства основной теоремы. Во второй части доказательство обобщается на множество всех натуральных n; кроме того, здесь при доказательстве применяются все необходимые для решения проблемы с полами и потолками математические методы.

В данном разделе асимптотические обозначения будут несколько видоизменены: с их помощью будет описываться поведение функций, определенных только для целых степеней числа b, хотя согласно определению асимптотических обозначений неравенства должны доказываться для всех достаточно больших чисел, а не только для степеней числа b. Поскольку можно ввести новые асимптотические обозначения, применимые к множеству чисел $\{b^i : i = 0, 1, 2, \ldots\}$, а не ко всему множеству неотрицательных целых чисел, упомянутое видоизменение несущественно.

Тем не менее, применяя асимптотические обозначения на суженной области определения, необходимо быть внимательным, чтобы не прийти к неверным выводам. Например, если доказано, что $T(n) = O(n)$, если n — степень двойки, то это еще не означает, что $T(n) = O(n)$ для всех n. Функция $T(n)$ может быть определена так.

$$T(n) = \begin{cases} n\,, & \text{если } n = 1, 2, 4, 8, \ldots\,, \\ n^2 & \text{в противном случае}\,. \end{cases}$$

В этом случае наилучшей верхней границей, как легко доказать, является $T(n) = O(n^2)$. Во избежание подобных ошибок мы никогда не будем использовать асимптотические обозначения на ограниченной области определения функции, если только из контекста не будет вполне очевидно, что именно мы собираемся делать.

4.6.1. Доказательство теоремы для точных степеней

В первой части доказательства основной теоремы анализируется рекуррентное соотношение (4.20),

$$T(n) = aT(n/b) + f(n) ,$$

в предположении, что n — точная степень числа $b > 1$, где b — необязательно целое число. Анализ проводится путем доказательства трех лемм. В первой из них решение основного рекуррентного соотношения сводится к вычислению выражения, содержащего суммирование. Во второй лемме определяются границы этой суммы. В третьей лемме с помощью первых двух доказывается версия основной теоремы для случая, когда n — точная степень b.

Лемма 4.2

Пусть $a \geq 1$ и $b > 1$ — константы, а $f(n)$ — неотрицательная функция, определенная для точных степеней числа b. Определим функцию $T(n)$ на множестве точных степеней числа b с помощью рекуррентного соотношения

$$T(n) = \begin{cases} \Theta(1) , & \text{если } n = 1 , \\ aT(n/b) + f(n) , & \text{если } n = b^i , \end{cases}$$

где i — положительное целое число. Тогда

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) . \tag{4.21}$$

Доказательство. Воспользуемся деревом рекурсии, представленным на рис. 4.7. Корень дерева имеет стоимость $f(n)$, и у него a дочерних ветвей, стоимость каждой из которых равна $f(n/b)$. (В ходе этих рассуждений, особенно для визуального представления дерева рекурсии, удобно считать число a целым, хотя это и не следует из каких-либо математических соотношений.) Каждая из этих дочерних ветвей, в свою очередь, также имеет a дочерних ветвей (таким образом, получается, что на втором уровне дерева имеется a^2 узлов), время выполнения каждой из которых равно $f(n/b^2)$. Обобщая эти рассуждения, можно сказать, что на j-м уровне находится a^j узлов, стоимость каждого из которых равна $f(n/b^j)$. Стоимость каждого листа равна $T(1) = \Theta(1)$, и все листья находятся на глубине $\log_b n$, поскольку $n/b^{\log_b n} = 1$. Всего же у дерева $a^{\log_b n} = n^{\log_b a}$ листьев.

Уравнение (4.21) можно получить, просуммировав стоимости всех листьев дерева, как показано на рисунке. Стоимость всех внутренних узлов уровня j равна $a^j f(n/b^j)$, так что полная стоимость всех внутренних узлов составляет

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) .$$

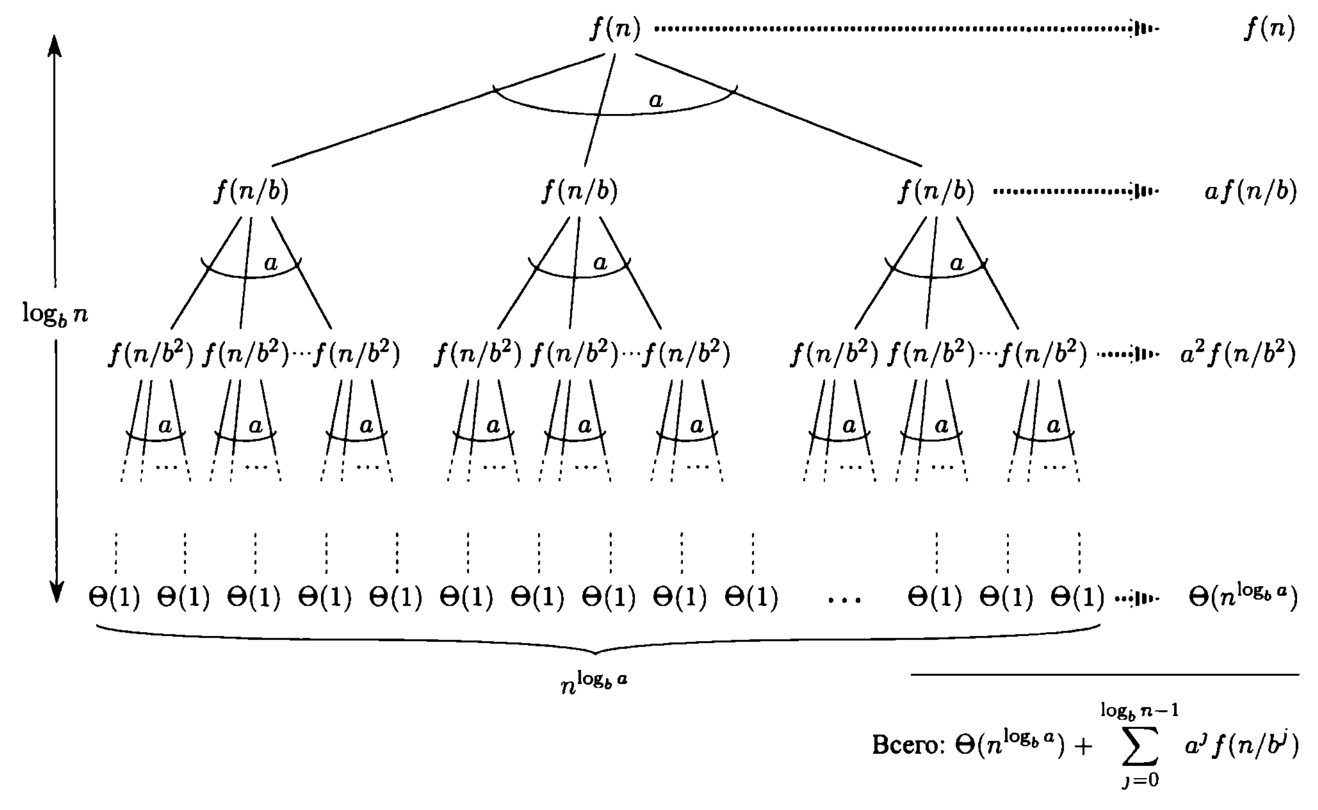


Рис. 4.7. Дерево рекурсии, генерируемое соотношением $T(n) = aT(n/b) + f(n)$. Это дерево представляет собой полное a-арное дерево с $n^{\log_b a}$ листьями и высотой $\log_b n$. Стоимость узлов на каждой глубине показана справа, а их сумма выражается уравнением (4.21).

В алгоритме "разделяй и властвуй", лежащем в основе анализируемого рекуррентного соотношения, эта сумма соответствует стоимости разделения поставленной задачи на вспомогательные подзадачи, и последующему объединению их решений. Стоимость всех листьев, т.е. стоимость решения всех $n^{\log_b a}$ вспомогательных подзадач размером 1, равна $\Theta(n^{\log_b a})$. ■

Три частных случая основной теоремы, выраженные в терминах дерева рекурсии, соответствуют ситуациям, когда полная стоимость дерева (1) преимущественно определяется стоимостью его листьев, (2) равномерно распределена по всем уровням дерева или (3) преимущественно определяется стоимостью корня.

Сумма в уравнении (4.21) описывает стоимость шагов разделения и комбинирования в алгоритме "разделяй и властвуй", лежащем в основе анализируемого рекуррентного соотношения. В следующей лемме обосновываются асимптотические оценки порядка роста этой суммы.

Лемма 4.3

Пусть $a \geq 1$ и $b > 1$ являются константами и пусть $f(n)$ представляет собой неотрицательную функцию, определенную для точных степеней b. Функция $g(n)$, определенная для точных степеней b с помощью соотношения

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) , \tag{4.22}$$

имеет следующие асимптотические границы для точных степеней b.

1. Если $f(n) = O(n^{\log_b a-\epsilon})$ для некоторой константы $\epsilon > 0$, то $g(n) = O(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, то $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Если $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$ и для всех достаточно больших n, то $g(n) = \Theta(f(n))$.

Доказательство. В случае 1 мы имеем $f(n) = O(n^{\log_b a-\epsilon})$, откуда следует, что $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$. Подстановка в уравнение (4.22) дает

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) . \quad (4.23)$$

Оценим сумму в O-обозначении. Для этого вынесем из-под знака суммирования постоянный множитель и выполним некоторые упрощения, в результате чего сумма преобразуется в возрастающую геометрическую прогрессию:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right) .
\end{aligned}$$

Поскольку b и ϵ — константы, можно переписать последнее выражение как $n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$. Подставив это выражение вместо суммы в уравнение (4.23), получим

$$g(n) = O(n^{\log_b a}) ,$$

тем самым доказав случай 1.

Поскольку в случае 2 предполагается, что $f(n) = \Theta(n^{\log_b a})$, получаем $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Подстановка в уравнение (4.22) приводит к

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \quad (4.24)$$

Ограничим сумму в Θ-обозначении, как в случае 1, но в этот раз геометрическую прогрессию мы не получим. Вместо этого мы обнаружим, что почти все члены

суммы одинаковы:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a} \log_b n \ .\end{aligned}$$

Подставив это выражение для суммы в уравнение (4.24), получаем

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n) \ ,\end{aligned}$$

что доказывает случай 2.

Случай 3 доказывается аналогично. Поскольку $f(n)$ фигурирует в определении (4.22) функции $g(n)$ и все члены $g(n)$ неотрицательны, можно заключить, что для точных степеней b справедливо соотношение $g(n) = \Omega(f(n))$. Согласно нашему предположению из формулировки леммы, для некоторой константы $c < 1$ и всех достаточно больших n выполняется неравенство $af(n/b) \leq cf(n)$. Переписав это предположение как $f(n/b) \leq (c/a)f(n)$ и выполнив j итераций, получим $f(n/b^j) \leq (c/a)^j f(n)$ или, что эквивалентно, $a^j f(n/b^j) \leq c^j f(n)$ в предположении, что итерируемые значения достаточно велики. Поскольку последнее (и наименьшее) такое значение равно n/b^{j-1}, хватит предположения о том, что n/b^{j-1} достаточно велико.

Подстановка в уравнение (4.22) и упрощение приводят к геометрической прогрессии, но в отличие от прогрессии в случае 1 в этот раз прогрессия убывающая. $O(1)$ используется для записи тех членов, которые не охватываются нашим предположением о том, что n достаточно велико,

$$\begin{aligned}g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \\ &\leq f(n) \sum_{j=0}^{\infty} c^j + O(1) \\ &= f(n)\left(\frac{1}{1-c}\right) + O(1) \\ &= O(f(n)) \ ,\end{aligned}$$

поскольку c является константой. Таким образом, можно сделать вывод, что $g(n) = \Theta(f(n))$ для точных степеней b. Доказав случай 3, мы завершили доказательство леммы. ■

Теперь можно доказать версию основной теоремы для случая, когда n представляет собой точную степень b.

Лемма 4.4

Пусть $a \geq 1$ и $b > 1$ — константы и пусть $f(n)$ — неотрицательная функция, определенная для точных степеней числа b. Определим функцию $T(n)$ для точных степеней числа b с помощью рекуррентного соотношения

$$T(n) = \begin{cases} \Theta(1) , & \text{если } n = 1 , \\ aT(n/b) + f(n) , & \text{если } n = b^i , \end{cases}$$

где i — положительное целое число. Тогда $T(n)$ имеет следующие асимптотические границы для точных степеней b.

1. Если $f(n) = O(n^{\log_b a - \epsilon})$ для некоторой константы $\epsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Если $f(n) = \Omega(n^{\log_b a + \epsilon})$ для некоторой константы $\epsilon > 0$ и если $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$ и всех достаточно больших n, то $T(n) = \Theta(f(n))$.

Доказательство. С помощью асимптотических границ из леммы 4.3 оценим значение суммы (4.21) из леммы 4.2. В случае 1 имеем

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) , \end{aligned}$$

а в случае 2

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) . \end{aligned}$$

В случае 3

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) , \end{aligned}$$

поскольку $f(n) = \Omega(n^{\log_b a + \epsilon})$. ■

4.6.2. Полы и потолки

Чтобы завершить доказательство основной теоремы, необходимо обобщить проведенный ранее анализ на случай, когда рекуррентное соотношение определе-

но не только для точных степеней числа b, но и для всех целых чисел. Получить нижнюю границу для выражения

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.25}$$

и верхнюю границу для

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.26}$$

не составляет труда, поскольку в первом случае для получения нужного результата можно использовать неравенство $\lceil n/b \rceil \geq n/b$, а во втором — неравенство $\lfloor n/b \rfloor \leq n/b$. Чтобы получить нижнюю границу рекуррентного соотношения (4.26), необходимо применить те же методы, что и при нахождении верхней границы для рекуррентного соотношения (4.25), поэтому здесь будет показан поиск только последней.

Дерево рекурсии, представленное на рис. 4.7, модифицируется и приобретает вид, показанный на рис. 4.8. По мере продвижения по дереву рекурсии вниз мы получаем следующую рекурсивную последовательность аргументов:

$$\begin{array}{l} n\,, \\ \lceil n/b \rceil\,, \\ \lceil \lceil n/b \rceil / b \rceil\,, \\ \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil\,, \\ \vdots \end{array}$$

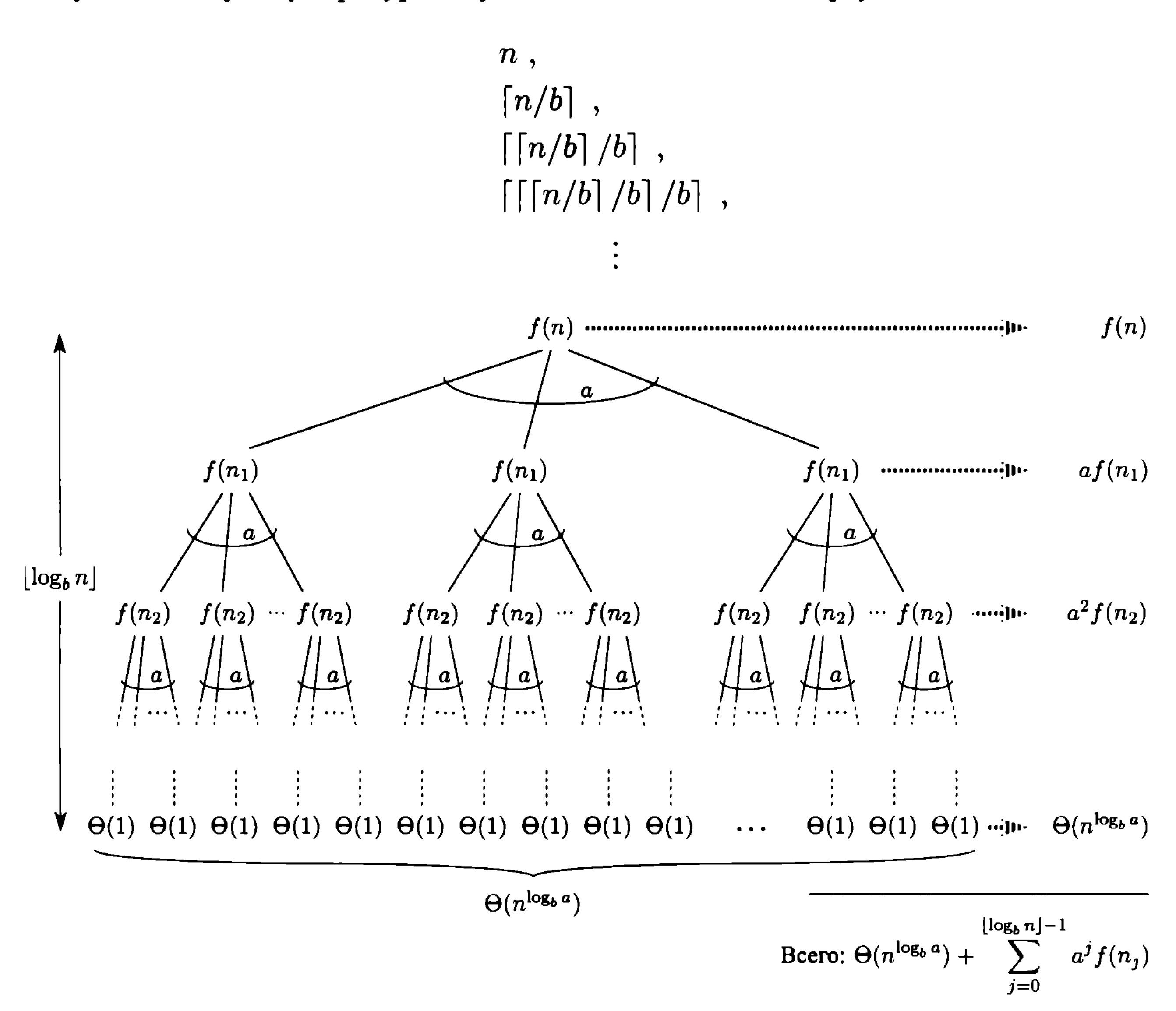


Рис. 4.8. Дерево рекурсии, генерируемое рекуррентным соотношением $T(n) = aT(\lceil n/b \rceil) + f(n)$. Аргумент рекурсии n_j определяется уравнением (4.27).

Обозначим j-й элемент этой последовательности как n_j, где

$$n_j = \begin{cases} n \;, & \text{если } j = 0 \;, \\ \lceil n_{j-1}/b \rceil \;, & \text{если } j > 0 \;. \end{cases} \tag{4.27}$$

Наша первая цель заключается в определении глубины k, такой, что n_k — константа. Воспользовавшись неравенством $\lceil x \rceil \leq x + 1$, получаем

$$\begin{aligned} n_0 &\leq n \;, \\ n_1 &\leq \frac{n}{b} + 1 \;, \\ n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1 \;, \\ n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 \;, \end{aligned}$$

В общем случае имеем

$$\begin{aligned} n_j &\leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} \;. \end{aligned}$$

Полагая $j = \lfloor \log_b n \rfloor$, получаем

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &< \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1) \;, \end{aligned}$$

и, таким образом, мы видим, что на глубине $\lfloor \log_b n \rfloor$ размер задачи не превышает константы.

Из рис. 4.8 видно, что

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) , \tag{4.28}$$

что почти то же самое, что и уравнение (4.21), с тем отличием, что n является произвольной целой константой, не ограниченной только точными степенями b.

Теперь можно вычислить сумму

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.29}$$

из уравнения (4.28) аналогично тому, как это делалось в доказательстве леммы 4.3. Начнем со случая 3. Если $af(\lceil n/b \rceil) \leq cf(n)$ для $n > b + b/(b-1)$, где $c < 1$ — константа, то $a^j f(n_j) \leq c^j f(n)$. Следовательно, сумму в уравнении (4.29) можно вычислять, как в лемме 4.3. В случае 2 имеем $f(n) = \Theta(n^{\log_b a})$. Если мы сможем показать, что $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, то доказательство случая 2 леммы 4.3 будет завершено. Заметим, что из $j \leq \lfloor \log_b n \rfloor$ следует $b^j/n \leq 1$. Наличие границы $f(n) = O(n^{\log_b a})$ подразумевает, что существует такая константа $c > 0$, что для всех достаточно больших n_j

$$\begin{aligned}
f(n_j) &\leq c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\
&\leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right) ,
\end{aligned}$$

поскольку $c(1+b/(b-1))^{\log_b a}$ представляет собой константу. Таким образом, случай 2 доказан. Доказательство случая 1 почти идентично. Ключевым моментом является доказательство границы $f(n_j) = O((n/b^j)^{\log_b a - \epsilon})$, которое аналогично соответствующему доказательству случая 2, хотя алгебраические выкладки при этом оказываются несколько более сложными.

Итак, мы доказали соблюдение верхних границ в основной теореме для всех целых n. Соблюдение нижних границ доказывается аналогично.

Упражнения

4.6.1 ★

Приведите простое и точное выражение для n_j в уравнении (4.27) для случая, когда b — положительное целое число (а не произвольное действительное).

4.6.2 ★

Покажите, что если выполняется соотношение $f(n) = \Theta(n^{\log_b a} \lg^k n)$, где $k \geq 0$, то основное рекуррентное соотношение имеет решение $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Для простоты рассмотрите только случай точных степеней b.

4.6.3 ★

Покажите, что в случае 3 основной теоремы одно из условий излишнее в том смысле, что из условия регулярности $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$ следует, что существует константа $\epsilon > 0$, такая, что $f(n) = \Omega(n^{\log_b a+\epsilon})$.

Задачи

4.1. Примеры рекуррентных соотношений

Определите верхнюю и нижнюю асимптотические границы функции $T(n)$ для каждого из представленных ниже рекуррентных соотношений. Считаем, что $T(n)$ при $n \leq 2$ является константой. Попытайтесь сделать эту оценку как можно более точной и обоснуйте свой ответ.

а. $T(n) = 2T(n/2) + n^4$.

б. $T(n) = T(7n/10) + n$.

в. $T(n) = 16T(n/4) + n^2$.

г. $T(n) = 7T(n/3) + n^2$.

д. $T(n) = 7T(n/2) + n^2$.

е. $T(n) = 2T(n/4) + \sqrt{n}$.

ж. $T(n) = T(n-2) + n^2$.

4.2. Стоимости передачи параметров

В этой книге предполагается, что передача параметров при вызове процедуры занимает фиксированное время, даже если передается N-элементный массив. Для большинства вычислительных систем это предположение справедливо, поскольку передается не сам массив, а указатель на него. В данной задаче исследуются три стратегии передачи параметров.

1. Массив передается посредством указателя. Время $= \Theta(1)$.

2. Массив передается посредством копирования. Время $= \Theta(N)$, где N — размер массива.
3. Массив передается путем копирования только некоторого поддиапазона, к которому обращается вызываемая процедура. Время $= \Theta(q-p+1)$ при передаче подмассива $A[p\mathinner{.\,.}q]$.

а. Рассмотрите рекурсивный алгоритм бинарного поиска, предназначенный для нахождения числа в отсортированном массиве (см. упр. 2.3.5). Приведите рекуррентные соотношения, описывающие время бинарного поиска в наихудшем случае, если массивы передаются с помощью каждого из описанных выше методов, и дайте точные верхние границы решений этих рекуррентных соотношений. Пусть размер исходной задачи равен N, а размер подзадачи — n.

б. Выполните задание части (а) для алгоритма MERGE-SORT из раздела 2.3.1.

4.3. Другие примеры рекуррентных соотношений

Дайте верхнюю и нижнюю асимптотические оценки функции $T(n)$ в каждом из приведенных ниже рекуррентных соотношений. Предполагается, что $T(n)$ для достаточно малых значений n является постоянной величиной. Постарайтесь, чтобы оценки были как можно более точными и обоснуйте ответы.

а. $T(n) = 4T(n/3) + n\lg n$.

б. $T(n) = 3T(n/3) + n/\lg n$.

в. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

г. $T(n) = 3T(n/3 - 2) + n/2$.

д. $T(n) = 2T(n/2) + n/\lg n$.

е. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

ж. $T(n) = T(n-1) + 1/n$.

з. $T(n) = T(n-1) + \lg n$.

и. $T(n) = T(n-2) + 1/\lg n$.

к. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4.4. Числа Фибоначчи

В этой задаче раскрываются свойства чисел Фибоначчи, определенных с помощью рекуррентного соотношения (3.22). Воспользуемся для решения рекуррентного соотношения Фибоначчи методом производящих функций. Определим

производящую функцию (или ***формальный степенной ряд***) $\mathcal{F}$ как

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

где F_i представляет собой i-е число Фибоначчи.

а. Покажите, что $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

б. Покажите, что

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z} \right) ,\end{aligned}$$

где

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots$$

и

$$\widehat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803\ldots .$$

в. Покажите, что

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \widehat{\phi}^i) z^i .$$

г. Воспользуйтесь п. (в) для доказательства того, что $F_i = \phi^i/\sqrt{5}$ (округленное до ближайшего целого) для всех $i > 0$. (*Указание:* обратите внимание, что $|\widehat{\phi}| < 1$.)

4.5. Тестирование микросхем

В распоряжении профессора есть n предположительно идентичных микросхем ("чипов"), которые в принципе способны тестировать друг друга. В тестирующее приспособление за один раз можно поместить две микросхемы. При этом каждая микросхема тестирует соседнюю и выдает отчет о результатах тестирования. Исправная микросхема всегда выдает правильные результаты тестирования другой микросхемы, а результатам неисправной микросхемы доверять нельзя. Таким образом, возможны четыре варианта результатов тестирования, приведенные в таблице ниже.

Отчет A	Отчет B	Заключение
B исправна	A исправна	Либо обе исправны, либо обе неисправны
B исправна	A неисправна	Неисправна как минимум одна микросхема
B неисправна	A исправна	Неисправна как минимум одна микросхема
B неисправна	A неисправна	Неисправна как минимум одна микросхема

а. Покажите, что, если как минимум $n/2$ микросхем неисправны, профессор не сможет точно определить исправные микросхемы, какой бы стратегией попарных испытаний он не пользовался. (Предполагается, что неисправные микросхемы не договариваются между собой, чтобы обмануть профессора.)

б. Рассмотрим задачу о поиске одной исправной микросхемы среди n микросхем, если предполагается, что исправно более половины всех микросхем. Покажите, что $\lfloor n/2 \rfloor$ попарных тестирований достаточно для сведения этой задачи к подзадаче, размер которой приблизительно в два раза меньше.

в. Покажите, что можно найти все исправные микросхемы с помощью $\Theta(n)$ попарных тестирований в предположении, что исправны более половины микросхем. Сформулируйте и решите рекуррентное соотношение, описывающее количество тестирований.

4.6. Массивы Монжа

Массив A размером $m \times n$, состоящий из действительных чисел, является ***массивом Монжа*** (Monge array), если для всех i, j, k и l, таких, что $1 \leq i < k \leq m$ и $1 \leq j < l \leq n$, мы имеем

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j] .$$

Другими словами, при любом выборе двух строк и двух столбцов из массива Монжа (при этом элементы массива на пересечении выбранных строк и столбцов образуют прямоугольник) сумма элементов в левом верхнем и правом нижнем углах полученного прямоугольника не превышает сумму элементов в левом нижнем и правом верхнем его углах. Приведем пример массива Монжа.

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

а. Докажите, что массив является массивом Монжа тогда и только тогда, когда для всех $i = 1, 2, ..., m-1$ и $j = 1, 2, ..., n-1$ выполняется условие

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j] .$$

(*Указание:* для прямого доказательства воспользуйтесь методом математической индукции отдельно для строк и для столбцов.)

б. Приведенный ниже массив не является массивом Монжа. Измените в нем один элемент таким образом, чтобы он стал массивом Монжа. (*Указание:* воспользуйтесь п. (а).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

в. Пусть $f(i)$ представляет собой индекс столбца, содержащего крайний слева минимальный элемент в строке i. Докажите, что $f(1) \leq f(2) \leq \cdots \leq f(m)$ для любого массива Монжа размером $m \times n$.

г. Далее приведено описание алгоритма "разделяй и властвуй", предназначенного для вычисления крайнего слева минимального элемента в каждой строке, входящей в состав массива Монжа A размером $m \times n$.

> Постройте подматрицу A' матрицы A, состоящую из четных строк матрицы A. С помощью рекурсивной процедуры найдите в каждой строке матрицы A' крайний слева минимальный элемент. Затем найдите крайний слева минимальный элемент среди нечетных строк матрицы A.

Объясните, как найти крайний слева минимальный элемент среди нечетных строк матрицы A (предполагается, что крайний слева минимальный элемент среди четных столбцов этой матрицы известен) за время $O(m + n)$.

д. Запишите рекуррентное соотношение, описывающее время работы алгоритма из п. (г). Покажите, что его решение имеет вид $O(m + n \log m)$.

Заключительные замечания

Применение метода "разделяй и властвуй" для разработки алгоритмов датируется как минимум 1962 годом, когда вышла статья Карацубы (Karatsuba) и Офмана (Ofman) [193]. Однако в действительности он мог применяться намного ранее; согласно Хайдеману (Heideman), Джонсону (Johnson) и Баррасу (Burrus) [162], К.Ф. Гаусс (C.F. Gauss) разработал первый алгоритм быстрого Фурье-преобразования в 1805 году, причем в формулировке Гаусса задача разбивается на меньшие подзадачи, решения которых затем объединяются.

Задача поиска максимального подмассива в разделе 4.1 представляет собой вариацию задачи, изучавшейся Бентли (Bentley) [42, глава 7].

Алгоритм Штрассена [323] вызвал большой ажиотаж после его опубликования в 1969 году. До этого мало кто представлял себе возможность существования алгоритма, асимптотически более быстрого, чем базовая процедура SQUARE-MATRIX-MULTIPLY. С того времени асимптотическая верхняя граница для матричного умножения была улучшена. На сегодня наиболее эффективный асимптотически алгоритм для перемножения матриц размером $n \times n$, разработанный Копперсмитом (Coppersmith) и Виноградом (Winograd) [77], имеет время работы $O(n^{2.376})$. Наилучшей известной нижней границей, совершенно очевидно, является $\Omega(n^2)$ (очевидно, поскольку мы должны заполнить n^2 элементов произведения матриц).

С практической точки зрения алгоритм Штрассена часто неприменим по следующим причинам.

1. Постоянный множитель, скрытый во времени работы алгоритма $\Theta(n^{\lg 7})$, оказывается гораздо больше постоянного множителя во времени $\Theta(n^3)$ процедуры SQUARE-MATRIX-MULTIPLY.
2. В случае разреженных матриц имеются специализированные более эффективные методы умножения.
3. Алгоритм Штрассена не настолько численно устойчив, как простой алгоритм умножения матриц SQUARE-MATRIX-MULTIPLY. Другими словами, из-за ограниченной точности компьютерных вычислений с действительными числами в случае алгоритма Штрассена накапливаются бóльшие ошибки, чем при использовании алгоритма SQUARE-MATRIX-MULTIPLY.
4. Построение подматриц на каждом шаге рекурсии приводит к повышенному расходу памяти.

Последние две причины потеряли актуальность в 1990-х годах. Хайем (Higham) [166] показал, что отличия в численной устойчивости преувеличены; хотя алгоритм Штрассена для ряда приложений действительно слишком численно неустойчив, все же его устойчивости вполне достаточно для большого количества приложений. Бейли (Bailey), Ли (Lee) и Саймон (Simon) [31] рассмотрели методы снижения требующейся для работы алгоритма Штрассена памяти.

На практике реализации быстрого умножения плотных матриц с использованием алгоритма Штрассена имеют "точку пересечения" (в смысле размера перемножаемых матриц) с обычным алгоритмом умножения и переключаются на использование простого алгоритма при перемножении матриц с размером, меньшим точки пересечения. Точное значение точки пересечения сильно зависит от реализации и используемой вычислительной системы. Анализ, учитывающий количество операций, но игнорирующий кеширование и конвейерную обработку, дает для точки пересечения достаточно низкие оценки: $n = 8$ у Хайема (Higham) [166] и $n = 12$ у Хасс-Ледермана (Huss-Lederman) и др. [185]). Д'Альберто (D'Alberto) и Николау (Nicolau) [80] разработали адаптивную схему, которая определяет точку пересечения при инсталляции пакета программного обеспечения. В различ-

ных вычислительных системах эта точка находится в диапазоне от $n = 400$ до $n = 2150$, а для ряда систем они так и не смогли ее найти.

Рекуррентные соотношения изучались еще с 1202 года, со времен Л. Фибоначчи (L. Fibonacci). А. де Муавр (A. De Moivre) впервые использовал метод производящих функций (см. задачу 4.4) для решения рекуррентных соотношений. Основной метод был принят на вооружение после появления работы Бентли (Bentley), Хакена (Haken) и Сакса (Saxe) [43], в которой было предложено расширение метода, обоснованное в упр. 4.6.2. Кнут (Knuth) [208][1] и Лью (Liu) [236] показали, каким образом можно решать линейные рекуррентные соотношения с использованием производящих функций. Подробное обсуждение методов решения рекуррентных соотношений можно найти в книгах Пурдома (Purdom) и Брауна (Brown) [285], а также Грехема (Graham), Кнута и Паташника (Patashnik) [151][2].

Некоторые исследователи, в число которых входят Акра (Akra) и Баззи (Bazzi) [13], Роура (Roura) [297], Верма (Verma) [344] и Яп (Yap) [358], предложили методы решения для более общих рекуррентных соотношений для алгоритмов "разделяй и властвуй", чем решаемые основным методом. Ниже представлены результаты Акра и Баззи, усовершенствованные Лейтоном (Leighton) [227]. Их метод подходит для решения рекуррентных соотношений вида

$$T(x) = \begin{cases} \Theta(1) , & \text{если } 1 \le x \le x_0 , \\ \sum_{i=1}^{k} a_i T(b_i x) + f(x) , & \text{если } x > x_0 , \end{cases} \tag{4.30}$$

где

- $x \ge 1$ является действительным числом,
- x_0 представляет собой константу, такую, что $x_0 \ge 1/b_i$ и $x_0 \ge 1/(1 - b_i)$ для $i = 1, 2, \ldots, k$,
- a_i — положительная константа; $i = 1, 2, \ldots, k$,
- b_i — константа из диапазона $0 < b_i < 1$; $i = 1, 2, \ldots, k$,
- $k \ge 1$ — целочисленная константа,
- $f(x)$ представляет собой неотрицательную функцию, удовлетворяющую ***условию полиномиального роста***: существуют положительные константы c_1 и c_2, такие, что для всех $x \ge 1$ при $i = 1, 2, \ldots, k$ и для всех u, таких, что $b_i x \le u \le x$, мы имеем $c_1 f(x) \le f(u) \le c_2 f(x)$. (Если $|f'(x)|$ ограничена сверху некоторым полиномом от x, то $f(x)$ удовлетворяет условию полиномиального роста. Например, $f(x) = x^\alpha \lg^\beta x$ удовлетворяет этому условию при любых действительных константах α и β.)

---

[1]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

[2]Имеется русский перевод: Р. Грэхем, Д. Кнут, О. Паташник. *Конкретная математика. Математические основы информатики, 2-е изд.* — М.: И.Д. "Вильямс", 2010.

Хотя основной метод неприменим к такому рекуррентному соотношению, как $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, метод Акра–Баззи в состоянии его решить. Для решения рекуррентного соотношения (4.30) сначала находим такое единственное действительное число p, что $\sum_{i=1}^{k} a_i b_i^p = 1$. (Такое p всегда имеется.) Тогда решение рекуррентного соотношения имеет вид

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}}\,du\right)\right) .$$

Метод Акра–Баззи несколько сложен в использовании, но успешно служит для решения рекуррентных соотношений, моделирующих разделение задачи на подзадачи существенно разного размера. Основной метод проще в использовании, но применим только тогда, когда размеры подзадач одинаковы.

Глава 5. Вероятностный анализ и рандомизированные алгоритмы

Эта глава знакомит читателя с вероятностным анализом и рандомизированными алгоритмами. Тем, кто не знаком с основами теории вероятности, следует обратиться к приложению В, в котором представлен обзор этого материала. В данной книге мы будем возвращаться к вероятностному анализу и рандомизированным алгоритмам неоднократно.

5.1. Задача о найме

Предположим, предприниматель хочет взять на работу нового офис-менеджера. Сначала он попытался найти подходящую кандидатуру самостоятельно, но потерпел неудачу и поэтому решил обратиться в агентство по трудоустройству. Согласно договоренности агентство должно присылать работодателю по одному кандидату в день. Предприниматель проводит с каждым из этих кандидатов собеседование, после чего принимает окончательное решение, брать его на работу или нет. За каждого направленного ему кандидата предприниматель платит агентству небольшую сумму. Однако наем выбранного кандидата на работу фактически стоит дороже, поскольку при этом необходимо уволить офис-менеджера, работающего в данное время, а также уплатить агентству более значительную сумму за выбранного кандидата. Предприниматель пытается сделать так, чтобы должность всегда занимал самый достойный из кандидатов. Как только квалификация очередного претендента окажется выше квалификации офис-менеджера, который работает в данное время, этот офис-менеджер будет уволен, а его место займет более подходящий кандидат. Работодатель готов понести все необходимые расходы, но хочет оценить, во что обойдется ему подбор нового сотрудника.

В представленной ниже процедуре HIRE-ASSISTANT эта стратегия найма представлена в виде псевдокода. В ней предполагается, что кандидаты на должность офис-менеджера пронумерованы от 1 до n. Предполагается также, что после собеседования с i-м кандидатом есть возможность определить, является ли он лучшим, чем все предыдущие. В процессе инициализации процедура создает фиктивного кандидата номер 0, квалификация которого ниже квалификации всех остальных.

```
HIRE-ASSISTANT(n)
best = 0                 // Кандидат 0 — фиктивный
for i = 1 to n           // неквалифицированный кандидат
    Беседа с кандидатом i
    if кандидат i лучше кандидата best
        best = i
        Нанять кандидата i
```

Модель стоимости этой задачи отличается от модели, описанной в главе 2. Если ранее в качестве стоимости алгоритма HIRE-ASSISTANT мы рассматривали бы время его работы, то теперь нас интересует денежная сумма, затрачиваемая на собеседование и наем при использовании данного алгоритма найма работника. На первый взгляд может показаться, что анализ стоимости этого алгоритма очень сильно отличается, например, от анализа времени работы алгоритма сортировки слиянием. Однако оказывается, что в ходе анализа стоимости и времени выполнения применяются одинаковые аналитические методы. В обоих случаях подсчитывается, сколько раз выполняется та или иная операция.

Обозначим невысокую стоимость собеседования как c_i, а существенно более высокую стоимость найма — как c_h. Пусть m — количество нанятых сотрудников. Тогда полная стоимость, затраченная при работе этого алгоритма, равна $O(c_i n + c_h m)$. Независимо от того, сколько сотрудников было нанято, интервью нужно провести с n кандидатами, поэтому суммарная стоимость всех интервью является фиксированной и равна $c_i n$. Таким образом, нас интересует анализ величины стоимости найма $c_h m$, которая, как нетрудно понять, меняется при каждом выполнении алгоритма.

Этот сценарий служит моделью распространенной вычислительной парадигмы. Часто встречаются ситуации, когда нужно найти максимальное или минимальное значение последовательности, для чего каждый ее элемент сравнивается с текущим "претендентом" на звание подходящего. Задача о найме моделирует частоту, с которой придется заново присваивать метку "победителя" текущему элементу.

Анализ наихудшего случая

В наихудшем случае работодателю приходится нанимать каждого кандидата, с которым проведено собеседование. Эта ситуация возникает, когда кандидаты приходят в порядке возрастания их квалификации. При этом процедура найма повторяется n раз, а полная стоимость найма равна $O(c_h n)$.

Однако маловероятно, чтобы кандидаты поступали в указанном порядке. Фактически мы не имеем представления о том, в каком порядке они будут приходить, и никак не можем повлиять на этот порядок. Таким образом, возникает закономерный вопрос, чего следует ожидать в типичном (или среднем) случае.

Вероятностный анализ

Вероятностный анализ — это анализ задачи, при котором используются вероятности тех или иных событий. Чаще всего он используется при определении времени работы алгоритма. Иногда такой анализ применяется и для оценки других величин, таких как стоимость найма в процедуре HIRE-ASSISTANT. Для проведения вероятностного анализа необходимо располагать знаниями (или сделать предположение) о распределении входных данных. При этом в ходе анализа алгоритма вычисляется время его работы в среднем случае путем усреднения времени работы по всем возможным входным данным. Вычисленное таким образом время работы называется ***временем работы в среднем случае***.

Делая предположение о распределении входных данных, нужно быть очень осторожным. В одних задачах такие предположения вполне оправданы и позволяют воспользоваться вероятностным анализом как методом разработки эффективных алгоритмов и как средством более глубокого понимания задачи. В других задачах разумно описать распределение входных величин не удается, и в таких случаях вероятностный анализ неприменим.

В задаче о найме сотрудника можно предположить, что претенденты приходят на собеседование в случайном порядке. Что это означает в контексте рассматриваемой задачи? Предполагается, что можно сравнить любых двух кандидатов и решить, кто из них квалифицированнее, т.е. в принципе всех кандидатов можно расположить в определенном порядке. (Определение полностью упорядоченных множеств приведено в приложении Б.) Таким образом, каждому кандидату можно присвоить уникальный ранг, используя числа от 1 до n. Обозначим ранг i-го претендента как $rank(i)$ и примем соглашение, что более высокий ранг соответствует специалисту более высокой квалификации. Упорядоченное множество $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ является перестановкой множества $\langle 1, 2, \ldots, n \rangle$. Утверждение, что кандидаты приходят на собеседование в случайном порядке, эквивалентно утверждению, что вероятность любого порядка рангов одинакова и равна количеству перестановок чисел от 1 до n, т.е. $n!$. Другими словами, ранги образуют ***случайную равновероятную перестановку***, т.е. каждая из $n!$ возможных перестановок появляется с одинаковой вероятностью.

Вероятностный анализ задачи о найме сотрудника приведен в разделе 5.2.

Рандомизированные алгоритмы

Чтобы провести вероятностный анализ, нужно иметь некоторые сведения о распределении входных данных. Во многих случаях мы знаем о них очень мало. Даже если об этом распределении что-то известно, может случиться так, что знания, которыми мы располагаем, нельзя численно смоделировать. Несмотря на это вероятность и случайность часто можно использовать в качестве инструмента в ходе разработки и анализа алгоритмов путем придания случайного характера части алгоритма.

В задаче о найме сотрудника все выглядит так, как будто кандидатов присылают на собеседование в случайном порядке, но у нас нет никакой возможности узнать, так ли это на самом деле. Поэтому, чтобы разработать рандомизированный

алгоритм для этой задачи, необходимо повысить степень контроля над порядком поступления претендентов на должность. Внесем в модель небольшие изменения. Допустим, бюро по трудоустройству подобрало n кандидатов. Работодатель договаривается, чтобы ему заранее прислали список кандидатов, и каждый день самостоятельно случайным образом выбирает, с кем из претендентов проводить собеседование. Несмотря на то что работодатель по-прежнему ничего не знает о претендентах на должность, задача существенно изменилась. Теперь не нужно гадать, будет ли очередность кандидатов случайной; вместо этого мы взяли этот процесс под свой контроль и сами сделали случайным порядок проведения собеседований.

В общем случае алгоритм называется ***рандомизированным*** (randomized), если его поведение определяется не только набором входных величин, но и значениями, которые выдает ***генератор случайных чисел***. Будем предполагать, что в нашем распоряжении имеется генератор дискретных случайных чисел RANDOM. Вызов RANDOM(a, b) возвращает целое число из интервала от a до b включительно, причем все числа равновероятны. Например, RANDOM$(0, 1)$ с вероятностью $1/2$ выдает 0 и с той же вероятностью — 1. В результате вызова RANDOM$(3, 7)$ каждое из чисел $3, 4, 5, 6$ и 7 возвращаются с вероятностью $1/5$. Вероятность возврата процедурой RANDOM любого целого числа не зависит от того, какие числа были возвращены предыдущим ее вызовом. Процедуру RANDOM можно представить в виде рулетки с $(b - a + 1)$ делениями. (На практике большинство сред программирования предоставляют в распоряжение программиста ***генератор псевдослучайных чисел***, т.е. детерминированный алгоритм, возвращающий числа, которые статистически "выглядят" случайными.)

В ходе анализа времени работы рандомизированного алгоритма мы получаем математическое ожидание времени работы для распределения значений, возвращаемых генератором случайных чисел. Мы отличаем такие алгоритмы от алгоритмов, в которых случайны входные данные, говоря о времени работы рандомизированного алгоритма как об ***ожидаемом времени работы***. В общем случае мы говорим о времени работы в среднем случае, когда случайным образом распределены входные данные алгоритма, и об ожидаемом времени работы, когда случайный выбор делает сам алгоритм.

Упражнения

5.1.1

Покажите, что из предположения о том, что в строке 4 процедуры HIRE-ASSISTANT всегда можно определить, какой кандидат наилучший, следует, что мы знаем общий порядок рангов кандидатов.

5.1.2 ★

Опишите реализацию процедуры RANDOM(a, b), которая может использовать только один вызов — процедуры RANDOM$(0, 1)$. Чему равно математическое ожидание времени работы вашей реализации как функции от a и b?

5.1.3 ★

Предположим, что нужно выводить 0 и 1 с вероятностью 1/2. В нашем распоряжении имеется процедура BIASED-RANDOM, которая с вероятностью p выдает 0 и с вероятностью $1 - p$ — число 1; значение p нам неизвестно. Сформулируйте алгоритм, использующий в качестве подпрограммы процедуру BIASED-RANDOM и возвращающий равномерно распределенные числа 0 и 1, т.е. вероятность вывода каждого из них равна 1/2. Чему равно математическое ожидание времени работы такой процедуры как функции от p?

5.2. Индикаторная случайная величина

В ходе анализа многих алгоритмов, в том числе того, с помощью которого решается задача о найме сотрудника, используется индикаторная случайная величина. Она предоставляет удобный метод перехода от вероятности к математическому ожиданию. Предположим, что в нашем распоряжении имеются пространство выборки S и событие A. Тогда ***индикаторная случайная величина*** (indicator random variable) $\mathrm{I}\{A\}$, связанная с событием A, определяется следующим образом:

$$\mathrm{I}\{A\} = \begin{cases} 1\ , & \text{если событие } A \text{ произошло}\ , \\ 0\ , & \text{если событие } A \text{ не произошло}\ . \end{cases} \tag{5.1}$$

В качестве простого примера определим математическое ожидание того, что при подбрасывании монеты выпадет орел. Пространство событий в этом случае имеет простой вид $S = \{H, T\}$, где вероятности выпадения орла (это событие обозначено как H (head)) и решки (T (tail)) равны: $\Pr\{H\} = \Pr\{T\} = 1/2$. Далее можно определить индикаторную случайную величину X_H, связанную с выпадением орла, т.е. с событием H. Эта величина подсчитывает количество выпадений орла. Если выпадает орел, она равна 1, а если решка — 0. Запишем это с помощью формальных обозначений:

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1\ , & \text{если выпал орел}\ , \\ 0\ , & \text{если выпала решка}\ . \end{cases} \end{aligned}$$

Математическое ожидание того, что выпадет орел, равняется математическому ожиданию индикаторной величины X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2\ . \end{aligned}$$

Таким образом, математическое ожидание того, что выпадет орел, равно $1/2$. Как показано в приведенной ниже лемме, математическое ожидание индикаторной случайной величины, связанной с событием A, равно вероятности этого события.

Лемма 5.1

Пусть имеются пространство событий S и событие A из пространства S и пусть $X_A = \mathrm{I}\{A\}$. Тогда $\mathrm{E}[X_A] = \Pr\{A\}$.

Доказательство. Согласно определению индикаторной случайной величины (5.1) и определению математического ожидания имеем:

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} \ , \end{aligned}$$

где $\overline{A}$ обозначает $S - A$, т.е. дополнение A. ■

Хотя индикаторные случайные величины могут показаться неудобными в применении, например при вычислении математического ожидания выпадений орла при бросании монеты, они полезны для анализа ситуаций, в которых делаются повторные испытания. Например, индикаторная случайная величина позволяет простым путем получить результат в уравнении (B.37). В этом уравнении количество выпадений орла при n бросках монеты вычисляется путем отдельного рассмотрения вероятности того события, что орел выпадет 0, 1, 2 и т.д. раз. В уравнении (B.38) предлагается более простой метод, в котором, по сути, неявно используются индикаторные случайные величины. Чтобы было понятнее, обозначим как X_i индикаторную случайную величину, связанную с событием, когда при i-м выбрасывании выпадает орел: $X_i = \mathrm{I}\{i\text{-й бросок приводит к событию } H\}$. Пусть X — случайная величина, равная общему количеству выпадений орла при n бросках монеты. Тогда

$$X = \sum_{i=1}^{n} X_i \ .$$

Необходимо вычислить математическое ожидание выпадения орла. Для этого применим операцию математического ожидания к обеим частям приведенного выше уравнения:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \ .$$

Приведенное выше уравнение дает нам математическое ожидание суммы n индикаторных случайных величин. Математическое ожидание каждой из этих случайных величин легко вычисляется с помощью леммы 5.1. Пользуясь уравнением (B.21), выражающим свойство линейности математического ожидания, математическое ожидание суммы случайных величин можно выразить как сумму математических ожиданий каждой величины. Благодаря линейности математи-

ческого ожидания использование индикаторной случайной величины становится мощным аналитическим методом, который применим даже в том случае, когда между случайными величинами имеется зависимость. Теперь мы можем легко вычислить математическое ожидание количества выпадений орла:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2 . \end{aligned}$$

Таким образом, индикаторные случайные величины значительно упростили вычисления по сравнению с методом, использующимся в уравнении (В.37). Вы еще не раз встретитесь с этими величинами в данной книге.

Анализ задачи о найме с помощью индикаторных случайных величин

Вернемся к задаче о найме сотрудника, в которой нужно вычислить математическое ожидание события, соответствующего найму нового менеджера. Чтобы провести вероятностный анализ, предположим, что кандидаты приходят на собеседование в случайном порядке (этот вопрос уже обсуждался в предыдущем разделе; в разделе 5.3 будет показано, каким образом можно обойтись без этого предположения). Пусть X — случайная величина, значение которой равно количеству наймов нового офис-менеджера. Далее можно воспользоваться определением математического ожидания (уравнение (В.20)) и получить соотношение

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

но эти вычисления окажутся слишком громоздкими. Вместо этого воспользуемся индикаторными случайными величинами, благодаря чему вычисления значительно упростятся.

Чтобы воспользоваться индикаторными случайными величинами, вместо вычисления величины $\mathrm{E}[X]$ путем определения одного значения, связанного с числом наймов нового менеджера, определим n величин, связанных с наймом конкретных кандидатов. В частности, пусть X_i — индикаторная случайная величина, связанная с событием найма i-го кандидата. Таким образом,

$$\begin{aligned} X_i &= \mathrm{I}\{\text{кандидат } i \text{ нанят}\} \\ &= \begin{cases} 1 \text{ , если кандидат } i \text{ нанят ,} \\ 0 \text{ , если кандидат } i \text{ не нанят} \end{cases} \end{aligned}$$

и

$$X = X_1 + X_2 + \cdots + X_n \ . \qquad (5.2)$$

В соответствии с леммой 5.1 имеем

$$\mathrm{E}[X_i] = \Pr\{\text{кандидат } i \text{ нанят}\} \ ,$$

и теперь нужно вычислить вероятность выполнения строк 5 и 6 процедуры HIRE-ASSISTANT.

Работодатель нанимает кандидата под номером i (строка 6), только если он оказывается лучше всех предыдущих (от 1-го до $i-1$-го). Поскольку мы предположили, что кандидаты приходят на собеседование в случайном порядке, это означает, что первые i кандидатов также прибыли в случайном порядке. Любой из первых i кандидатов с равной вероятностью может оказаться лучшим. Поэтому вероятность того, что квалификация претендента с номером i выше квалификации претендентов с номерами от 1 до $i-1$ и что он будет взят на работу, равна $1/i$. Пользуясь леммой 5.1, можно заключить, что

$$\mathrm{E}[X_i] = 1/i \ . \qquad (5.3)$$

Теперь можно вычислить $\mathrm{E}[X]$:

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(согласно (5.2))} && (5.4)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(в силу линейности математического ожидания)}\\
&= \sum_{i=1}^{n} 1/i && \text{(согласно (5.3))}\\
&= \ln n + O(1) && \text{(согласно (A.7))} \ . && (5.5)
\end{aligned}
$$

Даже если проведено интервью с n кандидатами, в среднем будет нанято только $\ln n$ из них. Этот результат подытожен в приведенной ниже лемме.

Лемма 5.2

В случае собеседования с кандидатами в случайном порядке полная стоимость найма при использовании алгоритма HIRE-ASSISTANT равна $O(c_h \ln n)$.

Доказательство. Эта граница следует непосредственно из нашего определения стоимости найма и уравнения (5.5), которое показывает, что математическое ожидание количества наймов приближенно равно $\ln n$. ∎

Стоимость найма в среднем случае существенно лучше, чем в наихудшем случае, когда она равна $O(c_h n)$.

Упражнения

5.2.1

Чему равна вероятность того, что в процедуре HIRE-ASSISTANT будет нанят ровно один кандидат, если предполагается, что собеседование с кандидатами проводится в случайном порядке? А вероятность того, что будут наняты n кандидатов?

5.2.2

Чему равна вероятность того, что в процедуре HIRE-ASSISTANT будет нанято ровно два кандидата, если предполагается, что собеседование с кандидатами проводится в случайном порядке?

5.2.3

Вычислите математическое ожидание суммы очков на n игральных костях с помощью индикаторных случайных величин.

5.2.4

Решите с помощью индикаторных случайных величин задачу, известную как ***задача о гардеробщике***. Каждый из n посетителей ресторана сдает свою шляпу в гардероб. Гардеробщик возвращает шляпы случайным образом. Чему равно математическое ожидание количества посетителей, получивших обратно свои собственные шляпы?

5.2.5

Пусть $A[1..n]$ — массив, состоящий из n различных чисел. Если $i < j$, а $A[i] > A[j]$, то пара (i, j) называется ***инверсией*** массива A (более детальную информацию об инверсиях можно найти в задаче 2.4). Предположим, что элементы массива A образуют равномерную случайную перестановку чисел $\langle 1, 2, \ldots, n \rangle$. Воспользуйтесь индикаторными случайными величинами для поиска математического ожидания количества инверсий в массиве.

5.3. Рандомизированные алгоритмы

В предыдущем разделе было показано, как знание информации о распределении входных данных может помочь проанализировать поведение алгоритма в среднем случае. Однако часто встречаются ситуации, когда такими знаниями мы не располагаем, и анализ среднего поведения невозможен. Но, как упоминалось в разделе 5.1, иногда есть возможность использовать рандомизированные алгоритмы.

В задачах наподобие задачи о найме, в которой важную роль играет предположение о равновероятности всех перестановок входных данных, вероятностный анализ приводит к разработке рандомизированного алгоритма. Вместо того чтобы предполагать то или иное распределение входных данных, мы попросту его навязываем. В частности, перед запуском алгоритма мы производим случайную

перестановку кандидатов, чтобы добиться выполнения условий равновероятности всех перестановок. Хотя при этом происходит модификация алгоритма, она не влияет на величину математического ожидания найма сотрудника, примерно равную $\ln n$. Но теперь мы можем ожидать это значение при *любых* входных данных независимо от их конкретного распределения.

Давайте рассмотрим отличия вероятностного анализа от рандомизированных алгоритмов. В разделе 5.2 мы выяснили, что если кандидаты приходят на собеседование в случайном порядке, то математическое ожидание количества наймов новых менеджеров приблизительно равно $\ln n$. Обратите внимание на то, что представленный алгоритм является детерминированным, т.е. для любых конкретных входных данных количество наймов новых менеджеров всегда будет одним и тем же. Кроме того, эта величина различна для разных входных данных и зависит от распределения рангов кандидатов. Поскольку количество наймов зависит только от рангов кандидатов, каждый отдельно взятый набор входных данных можно представить в виде перечисления рангов кандидатов в порядке нумерации последних, т.е. в виде последовательности $\langle rank(1), rank(2), \ldots, rank(n) \rangle$. В случае списка рангов $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$ новый менеджер будет всегда наниматься 10 раз, поскольку каждый очередной кандидат лучше предыдущего, и строки 5 и 6 будут выполняться при каждой итерации алгоритма. Если же список рангов имеет вид $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$, то новый менеджер будет нанят только один раз, во время первой итерации. Список $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$ дает три найма после интервью с кандидатами, имеющими ранг 5, 8 и 10. Напомним, что стоимость алгоритма зависит от того, сколько раз происходит наем нового сотрудника. Легко убедиться в том, что есть дорогостоящие входные данные (такие, как A_1), дешевые входные данные (такие, как A_2) и входные данные средней стоимости (такие, как A_3).

С другой стороны, рассмотрим теперь рандомизированный алгоритм, в котором сначала выполняется перестановка кандидатов на должность и только после этого определяется, кто из них самый лучший. В этом случае рандомизация представляет собой часть алгоритма, а не входных данных. При этом для отдельно взятого набора входных данных, например для представленного выше массива A_3, нельзя заранее сказать, сколько наймов будет выполнено, поскольку эта величина изменяется при каждом запуске алгоритма. При первом запуске алгоритма с входными данными A_3 в результате перестановки могут получиться входные данные A_1, и наем произойдет десять раз, а при втором запуске перестановка может дать входные данные A_2, и наем будет только один. Запустив алгоритм третий раз, можно получить еще какое-нибудь количество наймов. При каждом запуске алгоритма стоимость его работы зависит от случайного выбора и, скорее всего, будет отличаться от стоимости предыдущего запуска. В этом и во многих других рандомизированных алгоритмах *никакие входные данные не могут вызвать наихудшее поведение алгоритма*. Даже ваш злейший враг не сможет подобрать плохой входной массив, поскольку дальнейшая случайная перестановка приводит к тому, что порядок входных элементов становится несущественным. Рандомизированные алгоритмы плохо ведут себя лишь тогда, когда генератор случайных чисел выдает “неудачную” перестановку.

Единственное изменение, которое нужно внести в алгоритм найма для рандомизации, — это добавить код случайной перестановки.

RANDOMIZED-HIRE-ASSISTANT(n)

```
Случайная перестановка списка кандидатов
best = 0                // Кандидат 0 — фиктивный
for i = 1 to n          // неквалифицированный кандидат
    Беседа с кандидатом i
    if кандидат i лучше кандидата best
        best = i
        Нанять кандидата i
```

С помощью этого простого изменения создается рандомизированный алгоритм, производительность которого такая же, как и в случае предположения о том, что собеседование с кандидатами производится в случайном порядке.

Лемма 5.3

Математическое ожидание стоимости найма в процедуре RANDOMIZED-HIRE-ASSISTANT равно $O(c_h \ln n)$.

Доказательство. После перестановки элементов входного массива возникает ситуация, идентичная рассмотренной при вероятностном анализе алгоритма HIRE-ASSISTANT. ■

Сравнение лемм 5.2 и 5.3 выявляет различие между вероятностным анализом и рандомизированными алгоритмами. В лемме 5.2 делается предположение о виде входных данных. В лемме 5.3 такие предположения отсутствуют, хотя для рандомизации входных величин требуется дополнительное время. Для согласованности используемой терминологии в лемме 5.2 говорится о стоимости в среднем случае, а в лемме 5.3 — о математическом ожидании (ожидаемой) стоимости. В оставшейся части настоящего раздела обсуждаются некоторые вопросы, связанные с входными данными, которые подверглись случайной перестановке.

Массивы после случайной перестановки

Во многих алгоритмах входные данные рандомизируются путем перестановки элементов исходного входного массива (существуют и другие способы рандомизации). Мы рассмотрим два метода, позволяющие выполнить эту операцию. Будем считать, что у нас имеется массив A, который содержит элементы от 1 до n. Наша цель — получить случайную перестановку массива.

Один из распространенных методов заключается в том, чтобы присвоить каждому элементу $A[i]$ массива случайный приоритет $P[i]$, а затем отсортировать элементы массива A в соответствии с их приоритетами. Например, если исходный массив имеет вид $A = \langle 1, 2, 3, 4 \rangle$ и выбраны случайные приоритеты $P = \langle 36, 3, 62, 19 \rangle$, то в результате будет получен массив $B = \langle 2, 4, 1, 3 \rangle$, поскольку приоритет второго элемента самый низкий, за ним по приоритету идет

четвертый элемент, после него — первый и наконец — третий. Назовем эту процедуру PERMUTE-BY-SORTING.

```
PERMUTE-BY-SORTING(A)
1  n = A.length
2  Пусть P[1..n] — новый массив
3  for i = 1 to n
4      P[i] = RANDOM(1, n^3)
5  Отсортировать A, используя P в качестве ключей сортировки
```

В строке 4 выбирается случайное число от 1 до n^3. Такой интервал выбран для того, чтобы снизить вероятность наличия одинаковых приоритетов в массиве P. (В упр. 5.3.5 требуется доказать, что вероятность отсутствия в массиве одинаковых приоритетов составляет по меньшей мере $1-1/n$, а в упр. 5.3.6 — реализовать алгоритм, даже если несколько приоритетов могут иметь одинаковые значения.) Будем считать, что одинаковых приоритетов в массиве нет.

В представленном выше псевдокоде наиболее трудоемкая процедура — сортировка в строке 5. Как будет показано в главе 8, если использовать сортировку сравнением, то время ее работы составит $\Omega(n \lg n)$. Эта нижняя грань достижима, поскольку мы уже убедились, что сортировка слиянием выполняется в течение времени $\Theta(n \lg n)$. (В части II книги мы познакомимся и с другими видами сортировки, время работы которых также равно $\Theta(n \lg n)$. В упр. 8.3.4 предлагается решить очень похожую задачу сортировки чисел в диапазоне от 0 до $n^3 - 1$ за время $O(n)$.) Если после сортировки приоритет $P[i]$ является j-м в порядке возрастания, то элемент $A[i]$ на выходе алгоритма будет расположен в позиции с номером j. Таким образом, мы достигнем требуемой перестановки входных данных. Осталось доказать, что в результате выполнения этой процедуры будет получена ***случайная перестановка с равномерным распределением***, другими словами, что все перестановки чисел от 1 до n генерируются с одинаковой вероятностью.

Лемма 5.4

В предположении отсутствия одинаковых приоритетов в результате выполнения процедуры PERMUTE-BY-SORTING получается случайная перестановка входных значений с равномерным распределением.

Доказательство. Начнем с рассмотрения частной перестановки, в которой каждый элемент $A[i]$ получает i-й приоритет в порядке возрастания. Покажем, что вероятность такой перестановки равна $1/n!$. Обозначим через E_i ($i = 1, 2, \ldots, n$) событие, состоящее в том, что элемент $A[i]$ получает i-й приоритет. Теперь вычислим вероятность того, что события E_i происходят для всех i. Эта вероятность равна

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} .$$

Воспользовавшись результатами упр. B.2.5, можно показать, что эта вероятность равна

$$\Pr\{E_1\} \cdot \Pr\{E_2 \mid E_1\} \cdot \Pr\{E_3 \mid E_2 \cap E_1\} \cdot \Pr\{E_4 \mid E_3 \cap E_2 \cap E_1\} \\ \cdots \Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} \cdots \Pr\{E_n \mid E_{n-1} \cap \cdots \cap E_1\} \ .$$

Мы имеем $\Pr\{E_1\} = 1/n$, поскольку это вероятность того, что приоритет выбранного наугад одного из n элементов окажется минимальным. Теперь заметим, что $\Pr\{E_2 \mid E_1\} = 1/(n-1)$, поскольку при условии, что приоритет элемента $A[1]$ минимальный, каждый из оставшихся $n-1$ элементов имеет одинаковые шансы располагать вторым наименьшим приоритетом. В общем случае для $i = 2, 3, \ldots, n$ выполняется соотношение $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} = 1/(n-i+1)$, поскольку при условии, что приоритеты элементов от $A[1]$ до $A[i-1]$ равны от 1 до $i-1$ (в порядке возрастания), каждый из оставшихся $n-(i-1)$ элементов имеет одинаковые шансы располагать i-м наименьшим приоритетом. Таким образом, справедливо следующее выражение:

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ = \frac{1}{n!} \ ,$$

и мы показали, что вероятность получения тождественной перестановки равна $1/n!$.

Это доказательство можно обобщить на случай произвольной перестановки приоритетов. Рассмотрим произвольную фиксированную перестановку $\sigma = \langle\sigma(1), \sigma(2), \ldots, \sigma(n)\rangle$ множества $\{1, 2, \ldots, n\}$. Обозначим как r_i ранг приоритета, присвоенного элементу $A[i]$, причем элемент с j-м приоритетом имеет ранг, равный j. Если мы определим E_i как событие, при котором элемент $A[i]$ получает $\sigma(i)$-й приоритет (т.е. $r_i = \sigma(i)$), то можно будет провести доказательство, аналогичное приведенному выше. Таким образом, при вычислении вероятности получения той или иной конкретной перестановки рассуждения и расчеты будут идентичными изложенным выше. Поэтому вероятность получения такой перестановки также равна $1/n!$. ■

Возможно, некоторые читатели сделают вывод, что для доказательства того, что все случайные перестановки распределены с равной вероятностью, достаточно показать, что для каждого элемента $A[i]$ вероятность оказаться в позиции j равна $1/n$. Выполнив упр. 5.3.4, можно убедиться, что это условие недостаточное.

Более предпочтительный метод получения случайной перестановки — перестановка элементов заданного массива "на месте". С помощью процедуры RANDOMIZE-IN-PLACE эту операцию можно выполнить за время $O(n)$. В ходе i-й итерации элемент $A[i]$ случайным образом выбирается из множества элементов от $A[i]$ до элемента $A[n]$, после чего в последующих итерациях этот элемент больше не изменяется.

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      Обменять A[i] и A[RANDOM(i, n)]
```

Покажем с помощью инварианта цикла, что в результате выполнения процедуры RANDOMIZE-IN-PLACE получаются случайные перестановки с равномерным распределением. Назовем ***k-перестановкой*** (размещением) данного n-элементного множества последовательность, состоящую из k элементов, выбранных среди n элементов исходного множества (см. приложение В). Всего имеется $n!/(n-k)!$ возможных k-перестановок.

Лемма 5.5

Процедура RANDOMIZE-IN-PLACE вычисляет равномерно распределенные перестановки.

Доказательство. Используем следующий инвариант цикла.

> Непосредственно перед i-й итерацией цикла **for** в строках 2 и 3 для каждой возможной $(i-1)$-перестановки n элементов вероятность того, что подмассив $A[1..i-1]$ содержит эту $(i-1)$-перестановку, равна $(n-i+1)!/n!$.

Необходимо показать, что это утверждение истинно перед первой итерацией цикла, что итерации сохраняют истинность инварианта и что оно позволяет показать корректность алгоритма по завершении цикла.

Инициализация. Рассмотрим ситуацию непосредственно перед первой итерацией цикла, так что $i = 1$. Согласно формулировке инварианта цикла вероятность нахождения каждого размещения из 0 элементов в подмассиве $A[1..0]$ равна $(n-i+1)!/n! = n!/n! = 1$. Подмассив $A[1..0]$ — пустой, а 0-размещение по определению не содержит ни одного элемента. Таким образом, подмассив $A[1..0]$ содержит любое 0-размещение с вероятностью 1, и инвариант цикла перед первой итерацией выполняется.

Сохранение. Мы считаем, что перед i-й итерацией вероятность того, что в подмассиве $A[1..i-1]$ содержится заданное размещение $i-1$ элементов, равна $(n-i+1)!/n!$. Теперь нужно показать, что после i-й итерации каждая из возможных i-перестановок может находиться в подмассиве $A[1..i]$ с вероятностью $(n-i)!/n!$. Тогда увеличение i на следующей итерации приведет к сохранению инварианта цикла.

Изучим i-ю итерацию. Рассмотрим некоторое конкретное размещение i элементов и обозначим его элементы как $\langle x_1, x_2, \ldots, x_i \rangle$. Это размещение состоит из размещения $i-1$ элементов $\langle x_1, \ldots, x_{i-1} \rangle$, за которым следует значение x_i, которое помещается в ходе выполнения алгоритма в элемент $A[i]$. Пусть E_1 обозначает событие, при котором в результате первых $i-1$ итераций в подмассиве $A[1..i-1]$ создается определенное размещение $i-1$ элементов $\langle x_1, \ldots, x_{i-1} \rangle$. Согласно инварианту цикла $\Pr\{E_1\} = (n-i+1)!/n!$.

Пусть теперь E_2 — событие, при котором в ходе i-й итерации в позицию $A[i]$ помещается элемент x_i. Размещение $\langle x_1, \ldots, x_i \rangle$ формируется в подмассиве $A[1..i]$ только при условии, что происходят оба события — и E_1, и E_2, — так что мы должны вычислить значение $\Pr\{E_2 \cap E_1\}$. Воспользовавшись уравнением (В.14), получаем

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \; .$$

Вероятность $\Pr\{E_2 \mid E_1\}$ равна $1/(n-i+1)$, поскольку в строке 3 алгоритм выбирает x_i случайным образом среди $n-i+1$ значений в позициях $A[i..n]$. Таким образом, мы имеем

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} \; . \end{aligned}$$

Завершение. При завершении алгоритма $i = n+1$, и подмассив $A[1..n]$ представляет собой заданную n-перестановку с вероятностью $(n-(n+1)+1)!/n! = 0!/n! = 1/n!$.

Таким образом, процедура RANDOMIZE-IN-PLACE генерирует равномерно распределенные случайные перестановки. ■

Рандомизированный алгоритм зачастую является самым простым и наиболее эффективным средством решения задачи. Время от времени вы будете встречаться с такими алгоритмами в данной книге.

Упражнения

5.3.1

У профессора возникли возражения против инварианта цикла, использующегося при доказательстве леммы 5.5. Он сомневается, что этот инвариант выполняется перед первой итерацией. Согласно его доводам пустой подмассив не содержит никаких размещений из 0 элементов, поэтому вероятность того, что в таком подмассиве находится то или иное размещение, должна быть равна 0. Из этих рассуждений следует, что инвариант цикла перед первой итерацией не выполняется. Перепишите процедуру RANDOMIZE-IN-PLACE таким образом, чтобы связанный с ней инвариант цикла перед первой итерацией применялся к непустому подмассиву, и соответствующим образом модифицируйте доказательство леммы 5.5 для своей процедуры.

5.3.2

Профессор решил разработать алгоритм, в результате выполнения которого получались бы все случайные перестановки, кроме тождественной. Он предложил такую процедуру.

```
PERMUTE-WITHOUT-IDENTITY(A)
n = A.length
for i = 1 to n - 1
    Обменять A[i] с A[RANDOM(i + 1, n)]
```

Добьется ли профессор поставленной цели с помощью этого кода?

5.3.3

Предположим, что вместо того, чтобы менять местами элемент $A[i]$ со случайно выбранным элементом из подмассива $A[i \,..\, n]$, мы меняем его местами с любым случайно выбранным элементом массива A.

```
PERMUTE-WITH-ALL(A)
n = A.length
for i = 1 to n
    Обменять A[i] с A[RANDOM(1, n)]
```

Получится ли в результате выполнения этого кода равномерная случайная перестановка? Обоснуйте свой ответ.

5.3.4

Профессор предложил для генерации случайных перестановок с однородным распределением такую процедуру.

```
PERMUTE-BY-CYCLIC(A)
n = A.length
Пусть B[1..n] — новый массив
offset = RANDOM(1, n)
for i = 1 to n
    dest = i + offset
    if dest > n
        dest = dest - n
    B[dest] = A[i]
return B
```

Покажите, что каждый элемент $A[i]$ оказывается в определенной позиции массива B с вероятностью $1/n$. Затем покажите, что алгоритм профессора ошибочен в том смысле, что полученные в результате его выполнения перестановки не будут распределены равномерно.

5.3.5 ★

Докажите, что в результате выполнения процедуры PERMUTE-BY-SORTING вероятность отсутствия одинаковых элементов в массиве P не меньше величины $1 - 1/n$.

5.3.6

Объясните, как следует реализовать алгоритм PERMUTE-BY-SORTING в случае, когда два или более приоритетов идентичны. Другими словами, алгоритм должен выдавать случайные равномерно распределенные перестановки даже в случае, если два или более приоритетов имеют одинаковые значения.

5.3.7

Предположим, что мы хотим создать ***случайную выборку*** из множества $\{1, 2, 3, \ldots, n\}$, т.е. m-элементное подмножество S, где $0 \leq m \leq n$, такое, что каждое m-подмножество создается с одинаковой вероятностью. Один из способов состоит в присваивании $A[i] = i$ для $i = 1, 2, 3, \ldots, n$ и вызове RANDOMIZE-IN-PLACE(A), после чего следует просто взять первые m элементов массива. Этот метод выполняет n вызовов процедуры RANDOM. Если n гораздо меньше m, можно создать случайную выборку с помощью меньшего количества вызовов RANDOM. Покажите, что приведенная далее рекурсивная процедура возвращает случайное m-подмножество множества S, состоящего из элементов $\{1, 2, 3, \ldots, n\}$, причем каждое m-подмножество равновероятно, и при этом процедура RANDOM вызывается только m раз.

```
RANDOM-SAMPLE(m, n)
if m == 0
    return ∅
else S = RANDOM-SAMPLE(m − 1, n − 1)
    i = RANDOM(1, n)
    if i ∈ S
        S = S ∪ {n}
    else S = S ∪ {i}
    return S
```

★ 5.4. Вероятностный анализ и дальнейшее применение индикаторных случайных величин

В этом разделе повышенной сложности на четырех примерах иллюстрируется дальнейшее применение вероятностного анализа. В первом примере вычисляется вероятность того, что двое из k человек родились в один и тот же день года. Во втором примере рассматривается процесс случайного наполнения корзин шарами, в третьем — исследуется событие, при котором в процессе бросания монеты несколько раз подряд выпадает орел. В последнем примере анализируется раз-

новидность задачи о найме сотрудника, в которой решение о найме принимается без проведения собеседования со всеми кандидатами.

5.4.1. Парадокс дней рождения

Первым будет рассмотрен ***парадокс дней рождения***. Сколько людей нужно собрать в одной комнате, чтобы вероятность совпадения даты рождения у двух из них достигла 50%? Полученное в результате решения этой задачи число на удивление мало́. Парадокс в том, что это число намного меньше, чем количество дней в году.

Чтобы решить задачу, присвоим всем, кто находится в комнате, номера от 1 до k, где k — количество людей в комнате. Наличие високосных годов проигнорируем и предположим, что в каждом году $n = 365$ дней. Пусть для $i = 1, 2, \ldots, k$ величина b_i представляет собой дату, на которую приходится день рождения i-й персоны ($1 \leq b_i \leq n$). Предположим также, что дни рождения равномерно распределены по всему году, так что $\Pr\{b_i = r\} = 1/n$ для $i = 1, 2, \ldots, k$ и $r = 1, 2, \ldots, n$.

Вероятность того, что даты рождения двух человек i и j совпадают, зависит от того, является ли случайный выбор этих дат независимым. В дальнейшем предполагается, что дни рождения независимы, поэтому вероятность того, что i- и j-й посетители комнаты родились в день r, можно вычислить следующим образом:

$$\begin{aligned} \Pr\{b_i = r \text{ и } b_j = r\} &= \Pr\{b_i = r\} \Pr\{b_j = r\} \\ &= 1/n^2 \ . \end{aligned}$$

Таким образом, вероятность того, что оба они родились в один день, равна

$$\begin{aligned} \Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ и } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n \ . \end{aligned} \tag{5.6}$$

Интуитивно легче понять такое утверждение: если день рождения человека b_i зафиксирован, то вероятность того, что день рождения человека b_j выпадет на эту же дату, равна $1/n$. Таким образом, вероятность того, что даты рождения двух человек i и j совпадают, равна вероятности того, что день рождения одного из них выпадет на заданную дату. Однако обратите внимание, что это совпадение основано на предположении о независимости дней рождения.

Теперь можно проанализировать вероятность того, что хотя бы двое из k людей родились в один и тот же день, рассматривая дополняющее событие. Вероятность совпадения хотя бы двух дней рождения равна 1, уменьшенной на величину вероятности того, что все дни рождения различаются. Событие, при котором все

дни рождения различаются, можно представить так:

$$B_k = \bigcap_{i=1}^{k} A_i \ ,$$

где A_i — событие, состоящее в том, что день рождения i-го человека отличается от дня рождения j-го человека для всех $j < i$. Поскольку мы можем записать $B_k = A_k \cap B_{k-1}$, из уравнения (В.16) получим рекуррентное соотношение

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \ , \tag{5.7}$$

где в качестве начального условия выступает $\Pr\{B_1\} = \Pr\{A_1\} = 1$. Другими словами, вероятность того, что дни рождения $b_1, b_2, \ldots, b_k$ различны, равна произведению вероятности того, что различны дни рождения $b_1, b_2, \ldots, b_{k-1}$, на вероятность того, что $b_k \neq b_i$ при $i = 1, 2, \ldots, k-1$ при условии, что $b_1, b_2, \ldots, b_{k-1}$ различны.

Если $b_1, b_2, \ldots, b_{k-1}$ различны, условная вероятность того, что $b_k \neq b_i$ при $i = 1, 2, \ldots, k-1$, равна $\Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$, поскольку из общего количества n дней незанятыми остаются $n-(k-1)$ дней. Итеративно применив рекуррентное соотношение (5.7), получим

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\
&\ \ \vdots \\
&= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\cdots\Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right) \ .
\end{aligned}$$

Неравенство (3.12), $1 + x \leq e^x$, дает

$$\begin{aligned}
\Pr\{B_k\} &\leq e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\leq 1/2
\end{aligned}$$

при $-k(k-1)/2n \leq \ln(1/2)$. Вероятность того, что все k дней рождения различаются, не меньше величины $1/2$ при условии, что $k(k-1) \geq 2n\ln 2$. Решая квадратное уравнение относительно k, получим, что это условие выполняется при $k \geq (1 + \sqrt{1 + (8\ln 2)n})/2$, что при $n = 365$ дает значение $k \geq 23$. Таким образом, если в комнате не менее 23 человек, вероятность того, что хотя бы двое из них родились в один и тот же день, не меньше $1/2$. А, например, на Марсе год длится 669 дней, поэтому для того, чтобы добиться того же эффекта исхо-

дя из продолжительности марсианского года, понадобилось бы собрать не менее 31 марсианина.

Анализ с применением индикаторных случайных величин

Индикаторные случайные величины дают возможность проведения простого, хотя и приближенного, анализа парадокса дней рождения. Определим для каждой пары (i, j) находящихся в комнате k людей индикаторную случайную величину X_{ij}, $1 \leq i < j \leq k$ следующим образом:

$$\begin{aligned} X_{ij} &= \mathrm{I}\{\text{Дни рождения } i \text{ и } j \text{ совпадают}\} \\ &= \begin{cases} 1, & \text{если дни рождения } i \text{ и } j \text{ совпадают}, \\ 0 & \text{в противном случае}. \end{cases} \end{aligned}$$

Согласно (5.6) вероятность того, что у двух людей дни рождения совпадают, равна $1/n$, и, таким образом, согласно лемме 5.1 мы имеем

$$\begin{aligned} \mathrm{E}[X_{ij}] &= \Pr\{\text{Дни рождения } i \text{ и } j \text{ совпадают}\} \\ &= 1/n . \end{aligned}$$

Полагая X случайной величиной, которая представляет собой количество пар людей, дни рождения которых совпадают, имеем

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} .$$

Применив к обеим частям этого равенства операцию вычисления математического ожидания и воспользовавшись свойством ее линейности, получаем

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\ &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}] \\ &= \binom{k}{2}\frac{1}{n} \\ &= \frac{k(k-1)}{2n} . \end{aligned}$$

Поэтому, если $k(k-1) \geq 2n$, математическое ожидание количества пар людей, родившихся в один и тот же день, не меньше 1. Таким образом, если в комнате как минимум $\sqrt{2n} + 1$ людей, то можно ожидать, что хотя бы у двух из них дни рождения совпадают. При $n = 365$ и $k = 28$ математическое ожидание количества пар, родившихся в один и тот же день, равно $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$.

Таким образом, если в комнате находится 28 человек, то следует ожидать, что хотя бы у двух из них день рождения совпадет. На Марсе, где год длится 669 дней, для того чтобы добиться того же эффекта, понадобилось бы собрать не менее 38 марсиан.

В ходе первого анализа, в котором использовались только вероятности, определялось, сколько людей нужно собрать, чтобы вероятность существования пары с совпадающими днями рождения превысила $1/2$. В ходе второго анализа, в котором использовались индикаторные случайные величины, определялось количество людей, при котором математическое ожидание числа пар с одним днем рождения на двоих равно 1. Несмотря на то что в этих двух ситуациях точное количество людей различается, в асимптотическом пределе оно одинаково: $\Theta(\sqrt{n})$.

5.4.2. Шары и корзины

Рассмотрим процесс случайного наполнения b корзин одинаковыми шарами, пронумерованными натуральными числами от 1 до b. Шары опускаются в корзины независимо один от другого, и вероятность того, что шар окажется в некоторой из корзин, одинакова для всех корзин и равна $1/b$. Таким образом, процесс заполнения корзин шарами представляет собой последовательность испытаний по схеме Бернулли (см. раздел В.4) с вероятностью успеха $1/b$, где успех состоит в том, что шар попадает в заданную корзину. Эта модель может оказаться особенно полезной в ходе анализа хеширования (см. главу 11), и мы можем ответить на ряд интересных вопросов о процессе наполнения корзин шарами (в задаче В.1 вы встретитесь и с другими вопросами о шарах и корзинах).

Сколько шаров попадает в определенную корзину? Количество шаров, попавших в определенную корзину, подчиняется биномиальному распределению $b(k; n, 1/b)$. Если всего в корзины было опущено n шаров, то согласно уравнению (В.37) математическое ожидание количества шаров в корзине равно n/b.

Сколько в среднем требуется шаров для того, чтобы в данной корзине оказался один шар? Количество шаров, которое требуется для того, чтобы в данной корзине оказался шар, подчиняется геометрическому распределению с вероятностью $1/b$ и согласно уравнению (В.32) математическое ожидание количества шаров, которое следует опустить в корзины, равно $1/(1/b) = b$.

Сколько шаров нужно опустить в корзины для того, чтобы в каждой из них оказалось хотя бы по одному шару? Назовем “попаданием” опускание шара в пустую корзину. Нужно определить математическое ожидание количества опусканий n, необходимых для b попаданий.

С помощью попаданий n опусканий можно разбить на этапы. Этап под номером i длится от $(i-1)$-го попадания до i-го попадания. Первый этап состоит из одного (первого) опускания, поскольку если все корзины пусты, то мы с необходимостью получим попадание. На i-м этапе имеется $i-1$ корзин с шарами и $b-i+1$ пустых корзин. Таким образом, при каждом опускании шара на i-м этапе вероятность попадания равна $(b-i+1)/b$.

Пусть n_i — количество опусканий на i-м этапе. Тогда количество шаров, необходимых для попадания в b корзин, равно $n = \sum_{i=1}^{b} n_i$. Все случайные зна-

чения n_i подчиняются геометрическому распределению с вероятностью успеха $(b - i + 1)/b$, и в соответствии с уравнением (В.32) мы имеем

$$\mathrm{E}[n_i] = \frac{b}{b - i + 1} .$$

Пользуясь линейностью математического ожидания, получаем

$$\begin{aligned} \mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\ &= \sum_{i=1}^{b} \mathrm{E}[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b - i + 1} \\ &= b \sum_{i=1}^{b} \frac{1}{i} \\ &= b(\ln b + O(1)) \quad \text{(согласно (А.7))} . \end{aligned}$$

Таким образом, после примерно $b \ln b$ опусканий шаров в корзины можно ожидать, что шар будет в каждой корзине. Эта задача известна также как ***задача сборщика купонов***, которая говорит о том, что человек, пытающийся собрать все b различных купонов, для успеха своего предприятия должен собрать около $b \ln b$ случайно находящихся купонов.

5.4.3. Последовательности выпадения орлов

Предположим, что правильная (т.е. выпадение орла и решки равновероятны) монета подбрасывается n раз. Какого количества последовательных выпадений орла можно ожидать? Как покажет последующий анализ, эта величина ведет себя как $\Theta(\lg n)$.

Докажем сначала, что математическое ожидание длины наибольшей последовательности орлов представляет собой $O(\lg n)$. Вероятность того, что при очередном подбрасывании выпадет орел, равна $1/2$. Пусть A_{ik} — событие, когда последовательность выпадений орлов длиной не менее k начинается с i-го подбрасывания, или, более строго, A_{ik} — событие, когда при k последовательных подбрасываниях монеты $i, i + 1, \ldots, i + k - 1$ (где $1 \leq k \leq n$ и $1 \leq i \leq n - k + 1$) будут выпадать одни орлы. Поскольку подбрасывания монеты осуществляются независимо, для каждого данного события A_{ik} вероятность того, что во всех k подбрасываниях выпадут одни орлы, определяется следующим образом:

$$\Pr\{A_{ik}\} = 1/2^k . \tag{5.8}$$

6 Зак. 3726

Для $k = 2\lceil \lg n \rceil$

$$\begin{aligned}
\Pr\left\{A_{i,2\lceil \lg n \rceil}\right\} &= 1/2^{2\lceil \lg n \rceil} \\
&\leq 1/2^{2\lg n} \\
&= 1/n^2 ,
\end{aligned}$$

так что вероятность того, что последовательность повторных выпадений орлов длиной не менее $2\lceil \lg n \rceil$ начинается с i-го подбрасывания, довольно невелика. Имеется не более $n - 2\lceil \lg n \rceil + 1$ подбрасываний, с которых может начаться указанная последовательность орлов. Таким образом, вероятность того, что последовательность повторных выпадений орлов длиной не менее $2\lceil \lg n \rceil$ начинается при произвольном подбрасывании, равна

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\leq \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\
&< \sum_{i=1}^{n} 1/n^2 \\
&= 1/n . \qquad (5.9)
\end{aligned}$$

Справедливость этого соотношения следует из неравенства Буля (В.19), согласно которому вероятность объединения событий не превышает сумму вероятностей отдельных событий. (Заметим, что неравенство Буля выполняется даже для тех событий, которые не являются независимыми.)

Теперь воспользуемся неравенством (5.9) для ограничения длины самой длинной последовательности выпадения орлов. Пусть L_j ($j = 0, 1, 2, \ldots, n$) — событие, когда длина самой длинной последовательности выпадения орлов равна j. В соответствии с определением математического ожидания мы имеем

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} . \qquad (5.10)$$

Можно попытаться оценить эту сумму с помощью верхних границ каждой из величин $\Pr\{L_j\}$, аналогично тому, как это было сделано в неравенстве (5.9). К сожалению, этот метод не может обеспечить хороших оценок. Однако достаточно точную оценку можно получить с помощью некоторых интуитивных рассуждений, которые вытекают из проведенного выше анализа. Присмотревшись внимательнее, можно заметить, что в сумме (5.10) нет ни одного слагаемого, в котором оба множителя j и $\Pr\{L_j\}$ были бы большими. Почему? При $j \geq 2\lceil \lg n \rceil$ величина $\Pr\{L_j\}$ очень мала, а при $j < 2\lceil \lg n \rceil$ оказывается невелико само значение j. Выражаясь более формально, можно заметить, что события L_j для $j = 0, 1, \ldots, n$ несовместимы, поэтому вероятность того, что непрерывная последовательность выпадения орлов длиной не менее $2\lceil \lg n \rceil$ начинается с любого подбрасывания монеты, равна $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Согласно неравен-

ству (5.9) имеем $\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} < 1/n$. Кроме того, из $\sum_{j=0}^{n} \Pr\{L_j\} = 1$ вытекает $\sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} \leq 1$. Таким образом, получаем

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil\lg n\rceil-1} j \Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil\lg n\rceil-1} (2\lceil\lg n\rceil) \Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil\lg n\rceil \sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} + n \sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil\lg n\rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n) .
\end{aligned}$$

Вероятность того, что длина последовательности непрерывных выпадений орла превысит величину $r\lceil\lg n\rceil$, быстро убывает с ростом r. Для $r \geq 1$ вероятность того, что последовательность как минимум $r\lceil\lg n\rceil$ выпадений орлов начнется с i-го подбрасывания, равна

$$\begin{aligned}
\Pr\{A_{i,r\lceil\lg n\rceil}\} &= 1/2^{r\lceil\lg n\rceil} \\
&\leq 1/n^r .
\end{aligned}$$

Таким образом, вероятность образования непрерывной цепочки из последовательных выпадений орла, имеющей длину не менее $r\lceil\lg n\rceil$, не превышает $n/n^r = 1/n^{r-1}$. Это утверждение эквивалентно утверждению, что длина такой цепочки меньше величины $r\lceil\lg n\rceil$ с вероятностью не менее чем $1 - 1/n^{r-1}$.

В качестве примера рассмотрим серию из $n = 1000$ подбрасываний монеты. Вероятность того, что в этой серии орел последовательно выпадет не менее $2\lceil\lg n\rceil = 20$ раз, не превышает $1/n = 1/1000$. Вероятность непрерывного выпадения орла более $3\lceil\lg n\rceil = 30$ раз не превышает $1/n^2 = 1/1\,000\,000$.

Теперь давайте рассмотрим дополняющую нижнюю границу и докажем, что математическое ожидание длины самой длинной непрерывной последовательности выпадений орлов в серии из n подбрасываний равно $\Omega(\lg n)$. Чтобы доказать справедливость этого утверждения, разобьем серию из n подбрасываний приблизительно на n/s групп по s подбрасываний в каждой. Если выбрать $s = \lfloor(\lg n)/2\rfloor$, то можно показать, что с большой вероятностью по крайней мере в одной из этих групп окажутся все орлы, т.е. самая длинная последовательность выпадения орлов имеет длину как минимум $s = \Omega(\lg n)$. Затем мы покажем, что математическое ожидание длины такой последовательности равно $\Omega(\lg n)$.

Разобьем серию из n испытаний на несколько (не менее $\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$) групп из $\lfloor(\lg n)/2\rfloor$ последовательных подбрасываний. Оценим вероятность того, что в каждой из групп выпадет хотя бы по одной решке. Согласно уравнению (5.8) вероятность того, что в группе, которая начинается с i-го подбрасывания, выпадут все орлы, равна

$$\begin{aligned}\Pr\left\{A_{i,\lfloor(\lg n)/2\rfloor}\right\} &= 1/2^{\lfloor(\lg n)/2\rfloor} \\ &\geq 1/\sqrt{n}\ .\end{aligned}$$

Таким образом, вероятность того, что последовательность непрерывного выпадения орлов длиной не менее $\lfloor(\lg n)/2\rfloor$ не начинается с i-го подбрасывания, не превышает величину $1 - 1/\sqrt{n}$. Поскольку $\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$ групп образуются из взаимно исключающих независимых подбрасываний, вероятность того, что каждая такая группа *не* будет последовательностью выпадений орлов длиной $\lfloor(\lg n)/2\rfloor$, не превышает величину

$$\begin{aligned}(1-1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} &\leq (1-1/\sqrt{n})^{n/\lfloor(\lg n)/2\rfloor-1} \\ &\leq (1-1/\sqrt{n})^{2n/\lg n-1} \\ &\leq e^{-(2n/\lg n-1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n)\ .\end{aligned}$$

В приведенной выше цепочке соотношений были использованы неравенство (3.12), $1 + x \leq e^x$, и тот факт, что при достаточно больших n справедливо соотношение $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ (при желании вы можете в этом убедиться самостоятельно).

Таким образом, вероятность того, что длина самой большой последовательности выпадений орлов превосходит величину $\lfloor(\lg n)/2\rfloor$, равна

$$\sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \Pr\{L_j\} \geq 1 - O(1/n)\ . \tag{5.11}$$

Теперь можно вычислить нижнюю границу математического ожидания длины самой длинной последовательности орлов. Воспользовавшись в качестве отправной точки уравнением (5.10) и выполнив преобразования, аналогичные проведенным в ходе анализа верхней границы, получаем

$$\begin{aligned}\mathrm{E}[L] &= \sum_{j=0}^{n} j\Pr\{L_j\} \\ &= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} j\Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} j\Pr\{L_j\}\end{aligned}$$

$$\begin{aligned}
&\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\
&= 0 \cdot \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \Pr\{L_j\} \\
&\geq 0 + \lfloor(\lg n)/2\rfloor (1 - O(1/n)) \qquad \text{(согласно (5.11))} \\
&= \Omega(\lg n)\ .
\end{aligned}$$

Как и в случае парадокса дней рождений, более простой, но менее точный анализ можно провести с помощью индикаторных случайных величин. Пусть $X_{ik} = \mathrm{I}\{A_{ik}\}$ — индикаторная случайная величина, связанная с последовательным выпадением не менее k орлов, начиная с i-го подбрасывания монеты. Чтобы подсчитать общее количество таких последовательностей, определим

$$X = \sum_{i=1}^{n-k+1} X_{ik}\ .$$

Вычислив от обеих частей этого равенства математическое ожидание и использовав его линейность, получаем

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} 1/2^k \\
&= \frac{n-k+1}{2^k}\ .
\end{aligned}$$

Подставляя в полученное соотношение различные значения k, можно определить математическое ожидание количества последовательностей длиной k. Если это число окажется бóльшим (намного превышающим единицу), то следует ожидать большого количества последовательностей выпадения орлов длиной k с высокой вероятностью появления. Если же это число намного меньше единицы, то встретить такую последовательность в серии испытаний маловероятно. Если

$k = c \lg n$ для некоторой положительной константы c, получим

$$\begin{aligned} \mathrm{E}[X] &= \frac{n - c\lg n + 1}{2^{c\lg n}} \\ &= \frac{n - c\lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c\lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}) \ . \end{aligned}$$

Если число c достаточно велико, математическое ожидание количества последовательностей непрерывных выпадений орла длиной $c \lg n$ очень мало, из чего можно заключить, что это событие маловероятно. С другой стороны, если $c = 1/2$, то мы получаем $\mathrm{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, и можно ожидать, что будет немало последовательностей орлов длиной $(1/2) \lg n$. Поэтому вероятность того, что встретится хотя бы одна такая последовательность, достаточно велика. Исходя лишь из этих грубых оценок, можно заключить, что ожидаемая длина самой длинной последовательности орлов равна $\Theta(\lg n)$.

5.4.4. Задача о найме в оперативном режиме

В качестве последнего примера рассмотрим одну из разновидностей задачи о найме сотрудника. Предположим, что в целях выбора наиболее подходящего кандидата мы не хотим проводить собеседование со всеми претендентами. Мы не хотим также повторять процедуру оформления на работу нового сотрудника и увольнения старого в поисках наиболее подходящей кандидатуры. Вместо этого мы попытаемся подыскать такого кандидата, который максимально приблизится к наивысшей степени соответствия должности. При этом необходимо соблюдать одно условие: после каждого интервью нужно либо сразу предложить должность претенденту, либо отвергнуть его. Как достичь компромисса между количеством проведенных интервью и квалификацией взятого на работу кандидата?

Можно смоделировать эту задачу таким образом. После встречи с очередным кандидатом каждому из них можно дать оценку. Обозначим оценку i-го кандидата как $score(i)$ и предположим, что всем претендентам выставлены разные оценки. После встречи с j кандидатами будет известно, какой из этих j претендентов на должность получил максимальную оценку, однако остается неизвестным, найдется ли среди оставшихся $n-j$ кандидатов человек с более высокой квалификацией. Будем придерживаться следующей стратегии: выберем положительное целое число $k < n$, проведем интервью с k претендентами, отказав всем им в должности, а затем возьмем на работу первого из последующих претендентов, оценка которого будет превышать оценки всех предыдущих кандидатов. Если же самый квалифицированный специалист окажется среди первых k претендентов, то придется взять на работу n-го кандидата. Формальная реализация этой схемы представлена в приведенной ниже процедуре ON-LINE-MAXIMUM(k, n), которая возвращает номер нанимаемого кандидата.

```
ON-LINE-MAXIMUM(k, n)
1  bestscore = −∞
2  for i = 1 to k
3      if score(i) > bestscore
4          bestscore = score(i)
5  for i = k + 1 to n
6      if score(i) > bestscore
7          return i
8  return n
```

Мы хотим определить для каждого положительного k вероятность того, что будет нанят наиболее квалифицированный претендент. Затем выберем наилучшее из всех значений k и реализуем описанную стратегию с этим значением. Пока что полагаем k фиксированным. Обозначим наивысшую оценку кандидатов с номерами от 1 до j как $M(j) = \max_{1 \le i \le j} \{score(i)\}$. Пусть S — событие, определяемое как выбор самого квалифицированного кандидата, а S_i — событие, при котором самым квалифицированным нанятым на работу кандидатом оказался i-й. Поскольку все события S_i являются взаимоисключающими, выполняется соотношение $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Заметим, что поскольку согласно нашей стратеги ни один из первых k претендентов на работу не принимается, $\Pr\{S_i\} = 0$ для $i = 1, 2, \ldots, k$. Таким образом, получаем

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} . \tag{5.12}$$

Теперь вычислим величину $\Pr\{S_i\}$. Чтобы на работу был принят i-й кандидат, необходимо выполнение двух условий. Во-первых, в i-й позиции должен оказаться самый квалифицированный кандидат (обозначим это событие как B_i), а во-вторых, в ходе выполнения алгоритма не должен быть выбран ни один из претендентов, пребывающий на позициях с $k + 1$-й по $i - 1$-ю, что произойдет только тогда, когда при всех j, таких, что $k+1 \le j \le i-1$, в строке 6 будет выполняться условие $score(j) < bestscore$. (Поскольку оценки не повторяются, возможность равенства $score(j) = bestscore$ можно проигнорировать.) Другими словами, все оценки от $score(k+1)$ до $score(i-1)$ должны быть меньше $M(k)$; если же какая-то из них окажется больше $M(k)$, то будет возвращен индекс первой из оценок, превышающих все предыдущие. Обозначим как O_i событие, заключающееся в том, что на работу не взят ни один из претендентов, проходивших собеседование под номерами от $k + 1$ до $i - 1$. К счастью, события B_i и O_i независимы. Событие O_i зависит только от порядка нумерации кандидатов, которые находятся на позициях от 1 до $i - 1$, а событие B_i зависит только от того, превышает ли оценка кандидата i оценки всех прочих кандидатов. Порядок оценок в позициях от 1 до $i - 1$ не влияет на то, превышает ли оценка i-го претендента все предыдущие оценки, а квалификация i-го кандидата не влияет на расположение кандидатов с порядковыми номерами от 1 до $i - 1$. Таким образом, можно применить уравнение (B.15),

чтобы получить

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\} \ .$$

Ясно, что вероятность $\Pr\{B_i\}$ равна $1/n$, поскольку кандидат с наивысшей квалификацией может находиться в любой из n позиций с равной вероятностью. Чтобы произошло событие O_i, наиболее квалифицированный кандидат среди претендентов с номерами от 1 до $i-1$ должен находиться на одной из первых k позиций. Он с равной вероятностью может оказаться на любой из этих $i-1$ позиций, так что $\Pr\{O_i\} = k/(i-1)$ и, соответственно, $\Pr\{S_i\} = k/(n(i-1))$. Воспользовавшись уравнением (5.12), получим

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} \ .
\end{aligned}$$

Ограничим приведенную выше сумму сверху и снизу, заменив суммирование интегрированием. Согласно неравенствам (A.12) получаем

$$\int_k^n \frac{1}{x}\,dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\,dx \ .$$

Вычисление этих определенных интегралов дает границы

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)) \ ,$$

что приводит к достаточно точной оценке величины $\Pr\{S\}$. Поскольку нам нужно максимально повысить вероятность успешного исхода, постараемся выбрать значение k, при котором нижняя граница $\Pr\{S\}$ имеет максимум. (Выбор нижней границы продиктован еще и тем, что найти ее максимум легче, чем максимум верхней границы.) Дифференцируя выражение $(k/n)(\ln n - \ln k)$ по k, получаем

$$\frac{1}{n}(\ln n - \ln k - 1) \ .$$

Приравняв производную к нулю, найдем, что нижняя граница интересующей нас вероятности достигает максимального значения, когда $\ln k = \ln n - 1 = \ln(n/e)$ или, что то же самое, когда $k = n/e$. Таким образом, реализовав описанную выше

стратегию при $k = n/e$, мы наймем самого достойного кандидата с вероятностью, не меньшей $1/e$.

Упражнения

5.4.1

Сколько человек должно собраться в комнате, чтобы вероятность того, что день рождения у кого-нибудь из них совпадет с вашим, была не меньшей $1/2$? Сколько человек необходимо, чтобы вероятность того, что хотя бы двое из них родились 7 ноября, превысила величину $1/2$?

5.4.2

Предположим, что мы наполняем b корзин шарами до тех пор, пока в какой-то из корзин не окажется два шара. Все опускания шаров выполняются независимо, и шар с равной вероятностью может оказаться в любой корзине. Чему равно математическое ожидание количества опущенных в корзины шаров?

5.4.3 ★

В ходе анализа парадокса дней рождения было принято предположение о взаимной независимости всех дней рождения. Является ли это предположение существенным, или достаточно попарной независимости? Обоснуйте свой ответ.

5.4.4 ★

Сколько человек нужно пригласить на вечеринку, чтобы вероятность того, что *трое* из них родились в один и тот же день, достигла заметной величины?

5.4.5 ★

Какова вероятность того, что строка длиной k, составленная из символов n-элементного множества, является размещением k элементов этого множества? Как этот вопрос связан с парадоксом дней рождения?

5.4.6 ★

Предположим, что n шаров распределяются по n корзинам. Каждый шар опускается независимо от других и с равной вероятностью может оказаться в любой из корзин. Чему равно математическое ожидание количества пустых корзин? Чему равно математическое ожидание количества корзин с одним шаром?

5.4.7 ★

Уточните нижнюю оценку длины последовательности выпадений орлов. Для этого покажите, что при n подбрасываниях симметричной монеты вероятность того, что такая последовательность будет не длиннее $\lg n - 2 \lg \lg n$, меньше $1/n$.

Задачи

5.1. Вероятностный подсчет

С помощью b-битового счетчика можно вести подсчет до $2^b - 1$ элементов. Предложенный Р. Моррисом (R. Morris) ***вероятностный подсчет*** (probabilistic counting) позволяет проводить нумерацию намного большего количества элементов ценой потери точности.

Пусть значение переменной-счетчика $i = 0, 1, \ldots, 2^b - 1$ означает номер элемента n_i возрастающей последовательности неотрицательных чисел. Будем считать, что начальное значение счетчика равно нулю, т.е. $n_0 = 0$. Операция INCREMENT увеличивает значение счетчика i случайным образом. Если $i = 2^b - 1$, то в результате этого действия выдается сообщение о переполнении. В противном случае значение счетчика с вероятностью $1/(n_{i+1} - n_i)$ возрастает на единицу и остается неизменным с вероятностью $1 - 1/(n_{i+1} - n_i)$.

Если для всех $i \geq 0$ выбрать $n_i = i$, то мы получим обычный счетчик. Более интересная ситуация возникает, если выбрать, скажем, $n_i = 2^{i-1}$ для $i > 0$ или $n_i = F_i$ (i-е число Фибоначчи; см. раздел 3.2).

В задаче предполагается, что число n_{2^b-1} достаточно большое, чтобы вероятностью переполнения можно было пренебречь.

а. Покажите, что математическое ожидание значения счетчика после применения к нему n операций INCREMENT равно n.

б. Анализ дисперсии значения счетчика зависит от выбора последовательности n_i. Рассмотрим простой случай, когда $n_i = 100i$ для всех $i \geq 0$. Оцените дисперсию значения счетчика после выполнения операции INCREMENT n раз.

5.2. Поиск в неотсортированном массиве

В этой задаче исследуются три алгоритма поиска значения x в неотсортированном n-элементном массиве A.

Рассмотрим следующую рандомизированную стратегию: выбираем элемент массива A с произвольным индексом i и проверяем справедливость равенства $A[i] = x$. Если оно выполняется, то алгоритм завершается. В противном случае продолжаем поиск, случайным образом выбирая новые элементы массива A. Перебор индексов продолжается до тех пор, пока не будет найден такой индекс j, что $A[j] = x$, или пока не будут проверены все элементы массива. Заметим, что выбор каждый раз производится среди всех индексов массива, поэтому круг поиска не сужается и один и тот же элемент может проверяться неоднократно.

а. Напишите псевдокод процедуры RANDOM-SEARCH, реализующей описанную стратегию. Позаботьтесь, чтобы алгоритм прекращал работу после того, как будут проверены все индексы массива.

б. Предположим, что имеется ровно один индекс i, такой, что $A[i] = x$. Чему равно математическое ожидание количества индексов в массиве A, которые

будут проверены до того, как будет найден элемент x и процедура RANDOM-SEARCH завершит работу?

в. Обобщите решение сформулированной в п. (б) задачи для ситуации, когда имеются $k \geq 1$ индексов i, таких, что $A[i] = x$. Чему равно математическое ожидание количества индексов в массиве A, которые будут проверены до того, как будет найден элемент x и процедура RANDOM-SEARCH завершит работу? Ответ должен представлять собой функцию от величин n и k.

г. Предположим, что условие $A[i] = x$ не выполняется ни для какого индекса i. Чему равно математическое ожидание количества индексов в массиве A, которые придется проверить до того, как будут проверены все элементы массива и процедура RANDOM-SEARCH завершит работу?

Теперь рассмотрим детерминированный алгоритм линейного поиска DETERMINISTIC-SEARCH. В этом алгоритме поиск элемента x производится путем последовательной проверки элементов $A[1], A[2], A[3], \ldots, A[n]$ до тех пор, пока не произойдет одно из двух событий: либо будет найден элемент $A[i] = x$, либо будет достигнут конец массива. Предполагается, что все возможные перестановки элементов входного массива встречаются с одинаковой вероятностью.

д. Предположим, что имеется всего один индекс i, такой, что $A[i] = x$. Чему равно время работы процедуры DETERMINISTIC-SEARCH в среднем случае? Как ведет себя эта величина в наихудшем случае?

е. Обобщите решение сформулированной в п. (д) задачи для ситуации, когда имеются $k \geq 1$ индексов i, для которых $A[i] = x$. Чему равно время работы процедуры DETERMINISTIC-SEARCH в среднем случае? Как ведет себя эта величина в наихудшем случае? Ответ должен быть функцией от n и k.

ж. Предположим, что условие $A[i] = x$ не выполняется ни для какого элемента массива A. Чему равно время работы процедуры DETERMINISTIC-SEARCH в среднем случае? Как ведет себя эта величина в наихудшем случае?

Наконец рассмотрим рандомизированный алгоритм SCRAMBLE-SEARCH, в котором сначала выполняется случайная перестановка элементов входного массива, а затем в полученном массиве выполняется описанный выше линейный детерминированный поиск.

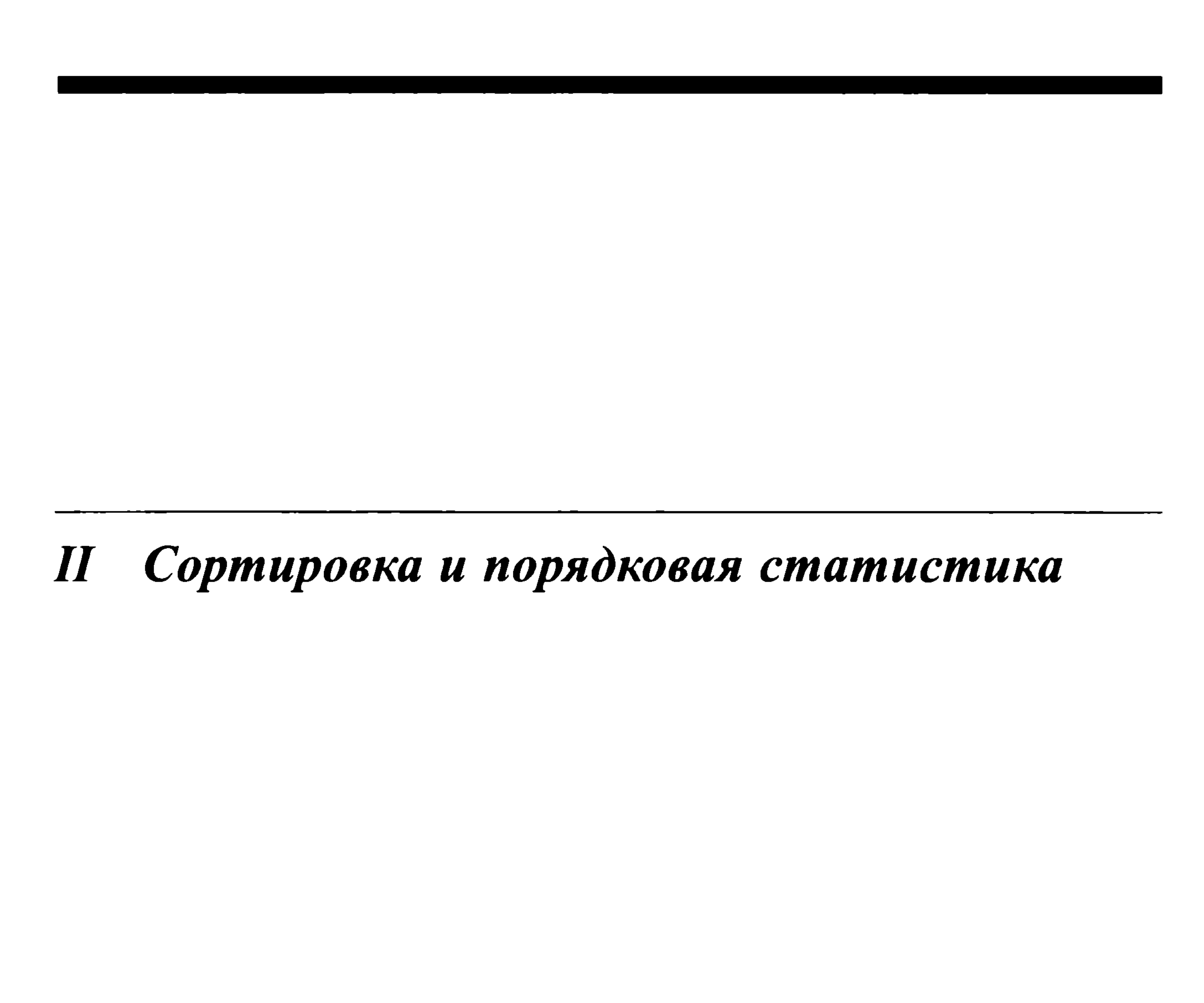
з. Пусть k — количество индексов i, таких, что $A[i] = x$. Определите математическое ожидание времени работы процедуры SCRAMBLE-SEARCH и время ее работы в наихудшем случае для значений $k = 0$ и $k = 1$. Обобщите решение для случая $k \geq 1$.

и. Какой из трех представленных алгоритмов вы бы предпочли? Поясните свой ответ.

Заключительные замечания

Описание большого количества усовершенствованных вероятностных методов можно найти в книгах Боллобаса (Bollobás) [52], Гофри (Hofri) [173] и лекциях Спенсера (Spencer) [319]. Обзор и обсуждение преимуществ рандомизированных алгоритмов представлен в работах Карпа (Karp) [199] и Рабина (Rabin) [286]. Кроме того, рандомизированные алгоритмы подробно рассмотрены в учебнике Мотвани (Motwani) и Рагхтвагана (Raghtvagan) [260].

Различные варианты задачи о найме изучались многими исследователями. Этот класс задач более известен как "задачи о секретарях". Примером работы в этой области является статья Айтаи (Ajtai), Меггидо (Meggido) и Ваартса (Waarts) [11].



II Сортировка и порядковая статистика

Введение

В этой части представлено несколько алгоритмов, с помощью которых можно решить следующую ***задачу сортировки***.

Вход. Последовательность из n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход. Перестановка (переупорядочение) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности, такая, что $a'_1 \le a'_2 \le \cdots \le a'_n$.

Входная последовательность обычно имеет вид n-элементного массива, хотя может иметь и другое представление, например в виде связанного списка.

Структура данных

На практике сортируемые числа редко являются изолированными значениями. Обычно каждое из них входит в состав набора данных, который называется ***записью*** (record). В каждой записи содержится ***ключ*** (key), представляющий собой сортируемое значение, в то время как остальная часть записи состоит из ***сопутствующих данных***, дополняющих ключ. Алгоритм сортировки на практике должен быть реализован так, чтобы он вместе с ключами переставлял и сопутствующие данные. Если каждая запись включает в себя сопутствующие данные большого объема, то с целью свести к минимуму перемещение данных сортировка часто проводится не в исходном массиве, а в массиве указателей на записи.

В определенном смысле сделанное выше замечание относится к особенностям реализации, отличающим алгоритм от конечной программы. Алгоритм сортировки описывает *метод* определения порядка сортировки вне зависимости от того, сортируются ли отдельные числа или большие записи с многобайтовыми сопутствующими данными. Таким образом, если речь идет о задаче сортировки, обычно предполагается, что входные данные состоят только из чисел. Преобразование алгоритма, предназначенного для сортировки чисел, в программу для сортировки записей не представляет концептуальных трудностей, хотя в конкретной

практической ситуации иногда могут возникнуть нюансы, усложняющие задачу программиста.

Почему сортировка?

Многие ученые в области вычислительной техники рассматривают сортировку как наиболее фундаментальную задачу при изучении алгоритмов. Тому есть несколько причин.

- Иногда в приложении не обойтись без сортировки информации. Например, чтобы подготовить отчет о состоянии счетов клиентов, банку необходимо выполнить сортировку чеков по их номерам.
- Часто в алгоритмах сортировка используется в качестве ключевой подпрограммы. Например, программе, выполняющей визуализацию перекрывающихся графических объектов, которые находятся на разных уровнях, сначала может понадобиться отсортировать эти объекты по уровням "снизу вверх", чтобы установить порядок их вывода. В этой книге мы ознакомимся с многочисленными алгоритмами, в которых сортировка используется в качестве подпрограммы.
- Имеется большой выбор алгоритмов сортировки, в которых применяются самые разные технологии. Фактически в алгоритмах сортировки используются многие важные методы (зачастую разработанные еще на заре компьютерной эры), применяемые при разработке различных классов алгоритмов. В этом отношении задача сортировки представляет также исторический интерес.
- Можно доказать наличие нетривиальной нижней границы для задачи сортировки (что будет сделано в главе 8). Наши наилучшие верхние границы в асимптотическом пределе совпадают с нижней границей, что дает возможность заключить, что наши алгоритмы сортировки являются асимптотически оптимальными. Кроме того, нижние оценки алгоритмов сортировки можно использовать для поиска нижних границ в некоторых других задачах.
- В процессе реализации алгоритмов сортировки на передний план выходят многие прикладные проблемы. Выбор наиболее производительной программы сортировки в той или иной ситуации может зависеть от многих факторов, таких как предварительные знания о ключах и сопутствующих данных, об иерархической организации памяти компьютера (наличии кеша и виртуальной памяти) и программной среды. Многие из этих вопросов лучше решать на уровне алгоритмов, а не "настройкой" кода.

Алгоритмы сортировки

В главе 2 мы познакомили вас с двумя алгоритмами для сортировки n действительных чисел. Сортировка методом вставок в наихудшем случае выполняется за время $\Theta(n^2)$. Несмотря на далеко не самое оптимальное асимптотическое поведение этого алгоритма, благодаря компактности его внутренних циклов он быстро справляется с сортировкой массивов с небольшим количеством элементов "на

месте" (без привлечения дополнительной памяти, т.е. без выделения отдельного массива для работы и хранения выходных данных). (Напомним, что выполнение сортировки ***на месте*** означает, что алгоритму требуется только определенное постоянное количество элементов вне исходного массива.) Сортировка методом слияния имеет асимптотически лучшее время работы $\Theta(n \lg n)$, но процедура MERGE, которая используется в этом алгоритме, не работает без дополнительной памяти.

В этой части вы ознакомитесь еще с двумя алгоритмами, предназначенными для сортировки произвольных действительных чисел. Представленная в главе 6 пирамидальная сортировка позволяет отсортировать на месте n чисел за время $O(n \lg n)$. В ней используется важная структура данных, именуемая пирамидой (heap), которая позволяет также реализовать очередь с приоритетами.

Рассмотренный в главе 7 алгоритм быстрой сортировки также сортирует n чисел "на месте", но время его работы в наихудшем случае равно $\Theta(n^2)$. Тем не менее его ожидаемое время работы — $\Theta(n \lg n)$, и на практике по производительности он превосходит алгоритм пирамидальной сортировки. Код алгоритма быстрой сортировки такой же компактный, как и код алгоритма сортировки вставкой, поэтому скрытый постоянный множитель, влияющий на величину времени работы этого алгоритма, довольно мал. Алгоритм быстрой сортировки приобрел широкую популярность для сортировки больших входных массивов.

Сортировки вставкой, слиянием, пирамидальная и быстрая имеют одну общую особенность — все они работают по принципу попарного сравнения элементов входного массива. В начале главы 8 рассматривается модель дерева принятия решения, позволяющая изучить ограничения производительности, присущие алгоритмам данного типа. С помощью этой модели доказывается, что в наихудшем случае нижняя оценка, ограничивающая время работы любого алгоритма, работающего методом сравнения, равна $\Omega(n \lg n)$. Это означает, что алгоритмы пирамидальной сортировки и сортировки слиянием являются асимптотически оптимальными.

Далее в главе 8 показано, что нижнюю границу $\Omega(n \lg n)$ можно превзойти, если информацию об отсортированном порядке входных элементов можно получить методами, отличными от попарного сравнения. Например, в алгоритме сортировки подсчетом предполагается, что входные данные образуют множество $\{0, 1, \ldots, k\}$. Используя в качестве инструмента для определения относительного порядка элементов массива механизм индексации, алгоритм сортировки перечислением может отсортировать n чисел за время $\Theta(k + n)$. Таким образом, при $k = O(n)$ время работы этого алгоритма линейно зависит от размера входного массива. Родственный алгоритм поразрядной сортировки может быть использован для расширения области применения сортировки перечислением. Если нужно выполнить сортировку n целых чисел, каждое из которых имеет d цифр, и при этом каждая цифра может принимать до k возможных значений, то алгоритм поразрядной сортировки справится с этой задачей за время $\Theta(d(n + k))$. Если d является константой, а величина k ведет себя, как $O(n)$, то время выполнения этого алгоритма линейно зависит от размера входного массива. Для применения третьего алгоритма — карманной сортировки — необходимы знания о распределе-

нии чисел во входном массиве. Этот алгоритм позволяет выполнить сортировку n равномерно распределенных на полуоткрытом интервале $[0, 1)$ действительных чисел за время $O(n)$ в среднем случае.

В следующей таблице содержится информация о времени работы алгоритмов сортировки из глав 2 и 6–8. Как обычно, n обозначает количество элементов, которые требуется отсортировать. В случае сортировки подсчетом сортируемыми элементами являются целые числа из множества $\{0, 1, \ldots, k\}$. В случае поразрядной сортировки каждый элемент представляет собой d-значное число, причем каждая его цифра принимает k различных значений. В карманной сортировке предполагается, что ключами являются равномерно распространенные на полуоткрытом интервале $[0, 1)$ действительные числа. В крайнем справа столбце приведено время работы алгоритма в среднем случае или ожидаемое время работы (с указанием вида времени работы в случае его отличия от времени работы в наихудшем случае). Время работы пирамидальной сортировки в среднем случае опущено, поскольку в данной книге оно не анализируется.

Алгоритм	Время работы в наихудшем случае	Время работы в среднем случае/ожидаемое
Сортировка вставкой	$\Theta(n^2)$	$\Theta(n^2)$
Сортировка слиянием	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Пирамидальная сортировка	$O(n \lg n)$	—
Быстрая сортировка	$\Theta(n^2)$	$\Theta(n \lg n)$ (ожидаемое)
Сортировка подсчетом	$\Theta(k + n)$	$\Theta(k + n)$
Поразрядная сортировка	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Карманная сортировка	$\Theta(n^2)$	$\Theta(n)$ (в среднем случае)

Порядковая статистика

В множестве, состоящем из n чисел, i-й порядковой статистикой называется i-е по величине значение в порядке возрастания. Конечно же, i-ю порядковую статистику можно выбрать путем сортировки входных элементов и индексирования i-го значения в выходных данных. Если не делается никаких предположений о распределении входных элементов, время работы данного метода равно $\Omega(n \lg n)$, как следует из величины нижней границы, найденной в главе 8.

В главе 9 будет показано, что i-й по величине (в порядке возрастания) элемент можно найти за время $O(n)$, даже если элементы представляют собой произвольные действительные числа. Мы представим рандомизированный алгоритм с компактным псевдокодом, время работы которого в наихудшем случае равно $\Theta(n^2)$, но при этом его ожидаемое время работы равно $O(n)$. Мы также приведем более сложный алгоритм со временем работы $O(n)$ в наихудшем случае.

Теоретическая подготовка

Хотя в основном в этой части не используется сложная математика, для освоения некоторых разделов требуется знание определенных разделов математики. В частности, в ходе анализа алгоритмов быстрой сортировки, карманной сортировки и алгоритма порядковой статистики используются некоторые положения теории вероятности, рассмотренные в приложении В, а также материал по вероятностному анализу и рандомизированным алгоритмам из главы 5. Анализ линейного алгоритма порядковой статистики в наихудшем случае содержит более сложные математические выкладки, чем используемые в ходе анализа наихудших случаев других рассмотренных в этой части алгоритмов.

Глава 6. Пирамидальная сортировка

В этой главе описывается еще один алгоритм сортировки, а именно — пирамидальная сортировка. Время работы этого алгоритма, как и время работы сортировки слиянием (и в отличие от времени работы сортировки вставкой), равно $O(n \lg n)$. Как и сортировка методом вставок, и в отличие от сортировки слиянием, пирамидальная сортировка выполняется без привлечения дополнительной памяти: в любой момент времени требуется память для хранения вне массива только некоторого постоянного количества элементов. Таким образом, в пирамидальной сортировке сочетаются наилучшие особенности двух рассмотренных ранее алгоритмов сортировки.

В ходе рассмотрения пирамидальной сортировки мы познакомимся с еще одним методом разработки алгоритмов, а именно — с использованием специализированных структур данных для управления информацией в ходе выполнения алгоритма. В рассматриваемом случае такая структура данных называется *пирамидой* (heap) и может оказаться полезной не только при пирамидальной сортировке, но и при создании эффективной очереди с приоритетами. В последующих главах эта структура данных появится снова.

Изначально термин "heap" использовался в контексте пирамидальной сортировки (heapsort), но позже его основной смысл изменился, и он стал обозначать память со сборкой мусора, в частности в языках программирования Lisp и Java (и переводиться как "куча"). Однако в данной книге термину "heap" (который здесь переводится как "пирамида") возвращен его первоначальный смысл.[1]

6.1. Пирамиды

Структура данных ***(бинарная) пирамида*** представляет собой объект-массив, который можно рассматривать как почти полное бинарное дерево (см. раздел Б.5.3), как показано на рис. 6.1. Каждый узел этого дерева соответствует

[1]Впрочем, поскольку перевод на русский язык термина *heap* в контексте структур данных и в контексте управления памятью разный, никаких проблем неоднозначности у русскоязычного читателя возникнуть не должно. — *Примеч. пер.*

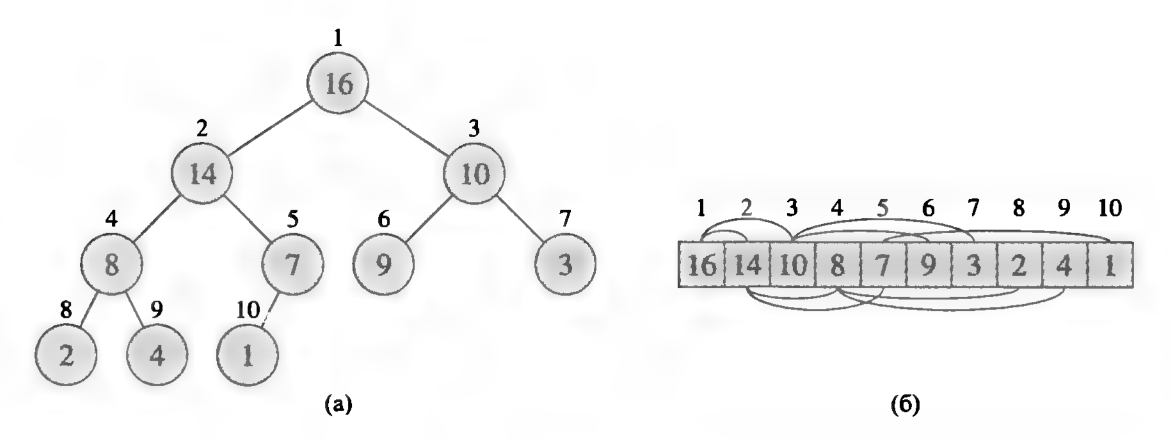


Рис. 6.1. Невозрастающая пирамида, представленная в виде **(а)** бинарного дерева и в виде **(б)** массива. Числа внутри кружков в каждом узле показывают значение, хранящееся в этом узле. Числа над узлами соответствуют индексам в массиве. Линии над и под элементами массива показывают отношения между родительскими и дочерними узлами; родительские узлы всегда находятся левее дочерних. Высота дерева равна трем; узел с индексом 4 (и значением 8) имеет высоту 1.

элементу массива. Дерево полностью заполнено на всех уровнях, за исключением, возможно, наинизшего, который заполняется слева направо. Массив A, представляющий пирамиду, является объектом с двумя атрибутами: $A.length$, который, как обычно, дает количество элементов в массиве, и $A.heap\text{-}size$, который указывает, сколько элементов пирамиды содержится в массиве A. Т.е. хотя $A[1\,..\,A.length]$ может содержать некоторые числа, только элементы подмассива $A[1\,..\,A.heap\text{-}size]$, где $0 \leq A.heap\text{-}size \leq A.length$, являются корректными элементами пирамиды. Корнем дерева является $A[1]$, а для заданного индекса i узла можно легко вычислить индексы его родительского, левого и правого дочерних узлов.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

На большинстве компьютеров операция $2i$ в процедуре LEFT выполняется с помощью одной команды процессора путем битового сдвига числа i на один бит влево. Аналогично операция $2i+1$ в процедуре RIGHT также выполняется очень быстро, путем сдвига бинарного представления числа i на одну позицию влево, а затем младший бит устанавливается равным 1. Процедура PARENT выполняется путем сдвига числа i на один бит вправо. Эффективные программы пирамидальной сортировки часто реализуют эти процедуры как макросы или встраиваемые процедуры.

Различают два вида бинарных пирамид: неубывающие и невозрастающие. В пирамидах обоих видов значения, расположенные в узлах, удовлетворяют ***свойству пирамиды*** (heap property), являющемуся отличительной чертой пирамиды

того или иного вида. ***Свойство невозрастающих пирамид*** (max-heap property) заключается в том, что для каждого отличного от корневого узла с индексом i выполняется неравенство

$$A[\text{PARENT}(i)] \geq A[i] ,$$

т.е. значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации ***неубывающей пирамиды*** (min-heap) прямо противоположный. ***Свойство неубывающих пирамид*** (min-heap property) заключается в том, что для всех отличных от корневого узлов с индексом i выполняется неравенство

$$A[\text{PARENT}(i)] \leq A[i] .$$

Таким образом, наименьший элемент такой пирамиды находится в ее корне.

В алгоритме пирамидальной сортировки используются невозрастающие пирамиды. Неубывающие пирамиды часто реализуют очереди с приоритетами (этот вопрос обсуждается в разделе 6.5). Для каждого приложения будет указано, с пирамидами какого вида мы будем иметь дело — с неубывающими или невозрастающими. При описании свойств, общих как для неубывающих, так и для невозрастающих пирамид, будет использоваться общий термин "пирамида".

Рассматривая пирамиду как дерево, определим ***высоту*** ее узла как число ребер в самом длинном простом нисходящем пути от этого узла к какому-то из листьев дерева. Высота пирамиды определяется как высота ее корня. Поскольку n-элементная пирамида строится по принципу полного бинарного дерева, ее высота равна $\Theta(\lg n)$ (см. упр. 6.1.2). Мы увидим, что время выполнения основных операций в пирамиде приблизительно пропорционально высоте дерева, и, таким образом, эти операции требуют для работы время $O(\lg n)$. В остальных разделах этой главы представлено несколько базовых процедур и продемонстрировано их использование в алгоритме сортировки и в структуре данных очереди с приоритетами.

- Процедура MAX-HEAPIFY выполняется за время $O(\lg n)$ и служит для поддержки свойства невозрастания пирамиды.
- Время выполнения процедуры BUILD-MAX-HEAP увеличивается с ростом количества элементов линейно. Эта процедура предназначена для создания невозрастающей пирамиды из неупорядоченного входного массива.
- Процедура HEAPSORT выполняется за время $O(n \lg n)$ и сортирует массив без привлечения дополнительной памяти.
- Процедуры MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY и HEAP-MAXIMUM выполняются за время $O(\lg n)$ и позволяют использовать пирамиду для реализации очереди с приоритетами.

Упражнения

6.1.1

Чему равно минимальное и максимальное количества элементов в пирамиде высотой h?

6.1.2

Покажите, что n-элементная пирамида имеет высоту $\lfloor \lg n \rfloor$.

6.1.3

Покажите, что в любом поддереве невозрастающей пирамиды корень этого поддерева содержит наибольшее значение среди узлов поддерева.

6.1.4

Где в невозрастающей пирамиде может находиться наименьший ее элемент, если все элементы различаются по величине?

6.1.5

Является ли массив с отсортированными элементами неубывающей пирамидой?

6.1.6

Является ли последовательность значений $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ невозрастающей пирамидой?

6.1.7

Покажите, что если n-элементную пирамиду представить в виде массива, то ее листьями будут элементы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2. Поддержка свойства пирамиды

Для поддержки свойства невозрастающей пирамиды мы вызываем процедуру MAX-HEAPIFY. Ее входными данными являются массив A и индекс i в этом массиве. При вызове процедуры MAX-HEAPIFY предполагается, что бинарные деревья с корнями LEFT(i) и RIGHT(i) представляют собой невозрастающие пирамиды, но $A[i]$ может быть меньше, чем значения в дочерних узлах (таким образом, нарушая свойство невозрастающей пирамиды). Процедура MAX-HEAPIFY "сплавляет" значение $A[i]$ вниз по невозрастающей пирамиде, так, что поддерево с корневым элементом с индексом i подчиняется свойству невозрастающей пирамиды.

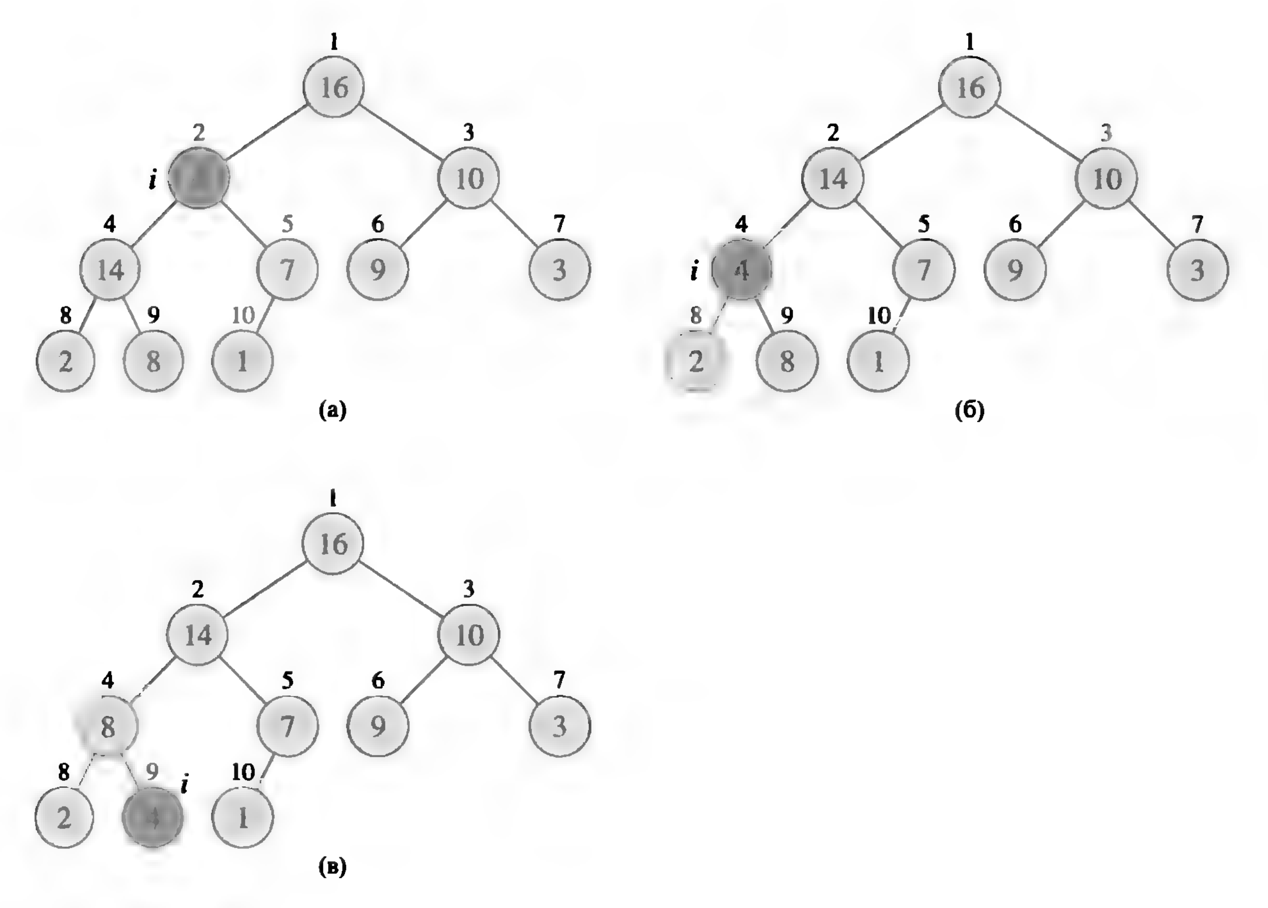


Рис. 6.2. Работа процедуры MAX-HEAPIFY(A, 2), где $A.heap\text{-}size = 10$. **(а)** Исходная конфигурация, в которой значение $A[2]$ в узле $i = 2$ нарушает свойство невозрастающей пирамиды, поскольку оно меньше, чем каждое из дочерних значений. Свойство невозрастающей пирамиды восстанавливается для узла 2 в части **(б)** путем обмена $A[2]$ с $A[4]$, который при этом нарушает свойство невозрастающей пирамиды для узла 4. В рекурсивном вызове MAX-HEAPIFY(A, 4) значение $i = 4$. После обмена $A[4]$ с $A[9]$, как показано в части **(в)**, ситуация в узле 4 исправляется, и рекурсивный вызов MAX-HEAPIFY(A, 9) не вносит никаких изменений в полученную структуру данных.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size и A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size и A[r] > A[largest]
    largest = r
if largest ≠ i
    Обменять A[i] и A[largest]
    MAX-HEAPIFY(A, largest)
```

На рис. 6.2 показана работа процедуры MAX-HEAPIFY. На каждом шаге определяется больший из элементов $A[i]$, $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, и его индекс сохраняется в переменной *largest*. Если наибольшим оказывается $A[i]$, то поддерево с корнем i уже представляет собой корректную невозрастающую пирамиду и процедура завершает работу. В противном случае наибольшим оказывается один из двух дочерних элементов, и процедура выполняет обмен $A[i]$ с $A[largest]$, что

приводит к тому, что для узла i и его дочерних узлов выполняется свойство невозрастающей пирамиды. Однако теперь исходное значение $A[i]$ оказывается в узле с индексом *largest*, так что поддерево с корнем *largest* может нарушать свойство невозрастающей пирамиды. Следовательно, необходимо рекурсивно вызвать процедуру MAX-HEAPIFY уже для этого дерева.

Для работы процедуры MAX-HEAPIFY на поддереве размером n с корнем в заданном узле i требуется время $\Theta(1)$, необходимое для исправления отношений между элементами $A[i]$, $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, плюс время работы этой процедуры с поддеревом, корень которого находится в одном из дочерних узлов узла i. Размер каждого из таких дочерних поддеревьев не превышает величину $2n/3$, причем наихудший случай осуществляется, когда последний уровень заполнен наполовину. Таким образом, время работы процедуры MAX-HEAPIFY описывается рекуррентным соотношением

$$T(n) \leq T(2n/3) + \Theta(1) .$$

Решение этого рекуррентного соотношения, согласно случаю 2 основной теоремы (теорема 4.1), имеет вид $T(n) = O(\lg n)$. По-другому время работы процедуры MAX-HEAPIFY с узлом, который находится на высоте h, можно выразить как $O(h)$.

Упражнения

6.2.1

Воспользовавшись рис. 6.2 в качестве образца, проиллюстрируйте работу процедуры MAX-HEAPIFY$(A, 3)$ с массивом $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2.2

Используя в качестве отправной точки процедуру MAX-HEAPIFY, напишите псевдокод процедуры MIN-HEAPIFY(A, i), которая выполняет соответствующие действия над неубывающей пирамидой. Каково время работы процедуры MIN-HEAPIFY по сравнению со временем работы процедуры MAX-HEAPIFY?

6.2.3

Как влияет на вызов процедуры MAX-HEAPIFY(A, i) ситуация, когда элемент $A[i]$ больше, чем его дочерние элементы?

6.2.4

К чему приведет вызов процедуры MAX-HEAPIFY(A, i) в случае $i > A.heap\text{-}size/2$?

6.2.5

Код процедуры MAX-HEAPIFY достаточно рационален, если не считать рекурсивного вызова в строке 10, из-за которого некоторые компиляторы могут сгенерировать неэффективный код. Напишите эффективную процедуру MAX-HEAPIFY, в которой вместо рекурсивного вызова использовалась бы итеративная управляющая конструкция (цикл).

6.2.6

Покажите, что в наихудшем случае время работы процедуры MAX-HEAPIFY на пирамиде размером n равно $\Omega(\lg n)$. (*Указание:* в пирамиде с n узлами присвойте узлам такие значения, чтобы процедура MAX-HEAPIFY рекурсивно вызывалась в каждом узле, расположенном на простом пути от корня до листа.)

6.3. Построение пирамиды

Процедуру MAX-HEAPIFY можно использовать в восходящем направлении для того, чтобы преобразовать массив $A[1\,..\,n]$, где $n = A.\mathit{length}$, в невозрастающую пирамиду. В соответствии с упр. 6.1.7 элементы в подмассиве $A[(\lfloor n/2 \rfloor + 1)\,..\,n]$ представляют собой листья дерева, поэтому каждый из них можно считать одноэлементной пирамидой, с которой можно начать процесс построения. Процедура BUILD-MAX-HEAP проходит по остальным узлам и для каждого из них выполняет процедуру MAX-HEAPIFY.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

На рис. 6.3 показан пример работы процедуры BUILD-MAX-HEAP.

Чтобы показать корректность работы процедуры BUILD-MAX-HEAP, воспользуемся следующим инвариантом цикла.

> В начале каждой итерации цикла **for** в строках 2 и 3 каждый узел node $i+1, i+2, \ldots, n$ является корнем невозрастающей пирамиды.

Необходимо показать, что этот инвариант справедлив перед первой итерацией цикла, сохраняется при каждой итерации и позволяет продемонстрировать корректность алгоритма после его завершения.

Инициализация. Перед первой итерацией цикла $i = \lfloor n/2 \rfloor$. Все узлы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ являются листьями, поэтому каждый из них является корнем тривиальной невозрастающей пирамиды.

Сохранение. Чтобы убедиться, что каждая итерация сохраняет инвариант цикла, заметим, что узлы, дочерние по отношению к узлу i, имеют номера, которые больше i. В соответствии с инвариантом цикла оба эти узла являются корнями невозрастающих пирамид. Это именно то условие, которое требуется для вызова процедуры MAX-HEAPIFY(A, i), чтобы преобразовать узел с индексом i в корень невозрастающей пирамиды. Кроме того, при вызове процедуры MAX-HEAPIFY сохраняется свойство пирамиды, заключающееся в том, что все узлы с индексами $i+1, i+2, \ldots, n$ являются корнями невозрастающих пирамид. Уменьшение индекса i в цикле **for** обеспечивает выполнение инварианта цикла для следующей итерации.

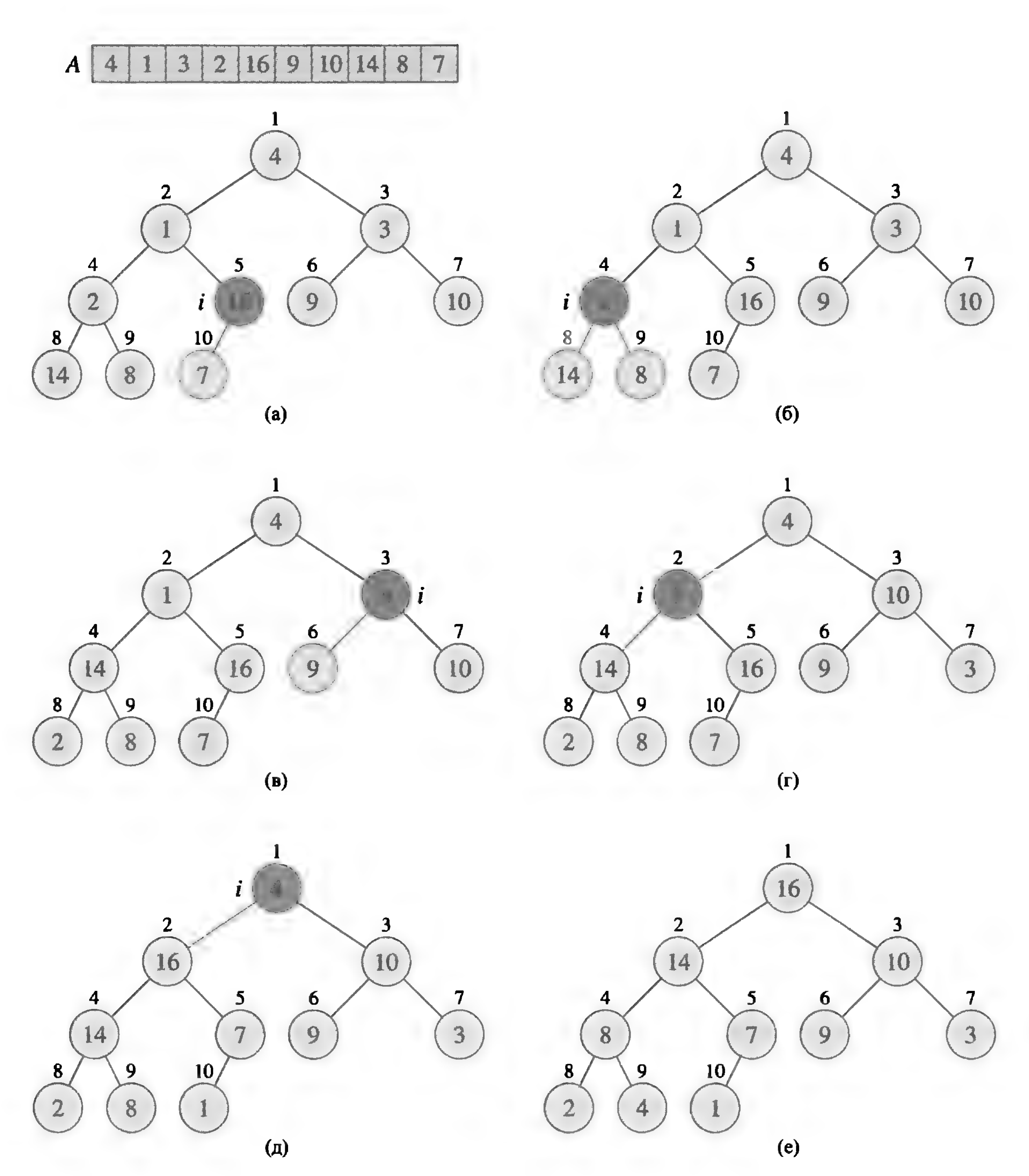


Рис. 6.3. Работа процедуры BUILD-MAX-HEAP. Показаны структуры данных перед вызовом MAX-HEAPIFY в строке 3 процедуры BUILD-MAX-HEAP. **(а)** 10-элементный входной массив A и бинарное дерево, которое он представляет. На рисунке показано, что индекс цикла i перед вызовом MAX-HEAPIFY(A, i) указывает на узел 5. **(б)** Полученная в результате структура данных. Индекс цикла i в следующей итерации указывает на узел 4. **(в)–(д)** Последовательные итерации цикла **for** в BUILD-MAX-HEAP. Обратите внимание, что, когда для некоторого узла вызывается процедура MAX-HEAPIFY, два поддерева этого узла представляют собой невозрастающие пирамиды. **(е)** Невозрастающая пирамида по завершении работы BUILD-MAX-HEAP.

Завершение. После завершения цикла $i = 0$. В соответствии с инвариантом цикла все узлы с индексами $1, 2, \dots, n$ являются корнями невозрастающих пирамид. В частности, таким корнем является узел 1.

Простую верхнюю оценку времени работы процедуры BUILD-MAX-HEAP можно получить следующим образом. Каждый вызов процедуры MAX-HEAPIFY имеет стоимость $O(\lg n)$, а всего имеется $O(n)$ таких вызовов. Таким образом, время работы алгоритма равно $O(n \lg n)$. Эта верхняя граница, будучи корректной, не является асимптотически точной.

Чтобы получить более точную оценку, заметим, что время работы MAX-HEAPIFY в том или ином узле зависит от высоты этого узла, и при этом большинство узлов расположено на малой высоте. При более тщательном анализе принимается во внимание тот факт, что высота n-элементной пирамиды равна $\lfloor \lg n \rfloor$ (упр. 6.1.2) и что на любом уровне на высоте h содержится не более $\lceil n/2^{h+1} \rceil$ узлов (упр. 6.3.3).

Время работы процедуры MAX-HEAPIFY при ее вызове для работы с узлом, который находится на высоте h, равно $O(h)$, так что общая стоимость процедуры BUILD-MAX-HEAP ограничена сверху значением

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

Последняя сумма вычисляется путем подстановки $x = 1/2$ в формулу (A.8), что дает

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2 . \end{aligned}$$

Таким образом, время работы процедуры BUILD-MAX-HEAP имеет верхнюю границу

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

Следовательно, построить невозрастающую пирамиду из неупорядоченного массива можно за линейное время.

Неубывающая пирамида строится с помощью процедуры BUILD-MIN-HEAP, идентичной процедуре BUILD-MAX-HEAP, лишь вызов MAX-HEAPIFY в строке 3 заменяется вызовом MIN-HEAPIFY (упр. 6.2.2). Процедура BUILD-MIN-HEAP строит неубывающую пирамиду из неупорядоченного массива за линейное время.

Упражнения

6.3.1

Воспользовавшись в качестве образца рис. 6.3, проиллюстрируйте работу процедуры Build-Max-Heap над входным массивом $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3.2

Почему индекс цикла i в строке 2 процедуры Build-Max-Heap убывает от $\lfloor A.length/2 \rfloor$ до 1, а не возрастает от 1 до $\lfloor A.length/2 \rfloor$?

6.3.3

Покажите, что в любой n-элементной пирамиде на высоте h находится не более $\lceil n/2^{h+1} \rceil$ узлов.

6.4. Алгоритм пирамидальной сортировки

Работа алгоритма пирамидальной сортировки начинается с вызова процедуры Build-Max-Heap для построения невозрастающей пирамиды из входного массива $A[1..n]$, где $n = A.length$. Поскольку наибольший элемент массива находится в корне $A[1]$, его можно поместить в корректную окончательную позицию в отсортированном массиве, поменяв его местами с элементом $A[n]$. Выбросив из пирамиды узел n (путем уменьшения на единицу величины $A.heap\text{-}size$, мы обнаружим, что дочерние поддеревья корня остаются корректными невозрастающими пирамидами, и только корень может нарушать свойство невозрастающей пирамиды. Для восстановления этого свойства достаточно вызвать процедуру Max-Heapify$(A, 1)$, после чего подмассив $A[1..n-1]$ превратится в невозрастающую пирамиду. Затем алгоритм пирамидальной сортировки повторяет описанный процесс для невозрастающих пирамид размером от $n-1$ до 2. (См. упр. 6.4.2, посвященное точной формулировке инварианта цикла данного алгоритма.)

```
Heapsort(A)
1  Build-Max-Heap(A)
2  for i = A.length downto 2
3      Обменять A[1] с A[i]
4      A.heap-size = A.heap-size − 1
5      Max-Heapify(A, 1)
```

На рис. 6.4 приведен пример работы процедуры Heapsort, после того как в строке 1 была построена начальная невозрастающая пирамида. На рисунке показана невозрастающая пирамида перед первой итерацией цикла **for** в строках 2–5 и после каждой из итераций.

Время работы процедуры Heapsort равно $O(n \lg n)$, поскольку вызов процедуры Build-Max-Heap требует времени $O(n)$, а каждый из $n - 1$ вызовов процедуры Max-Heapify — времени $O(\lg n)$.

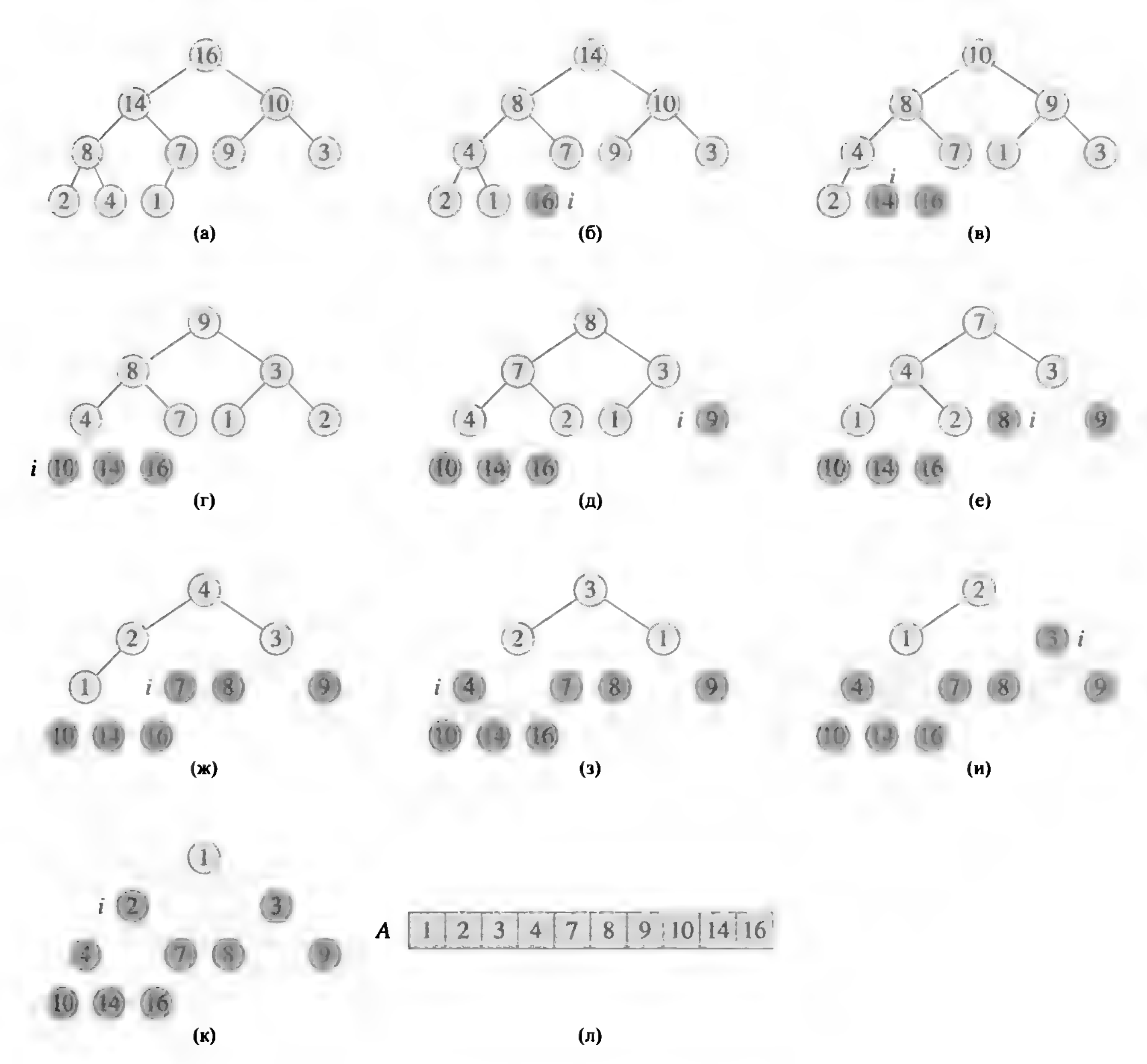


Рис. 6.4. Работа процедуры HEAPSORT. **(а)** Невозрастающая пирамида сразу после построения процедурой BUILD-MAX-HEAP в строке 1. **(б)–(к)** Невозрастающая пирамида после каждого вызова MAX-HEAPIFY в строке 5; указано также значение i в этот момент. В невозрастающей пирамиде остаются только светлые узлы. **(л)** Выходной отсортированный массив A.

Упражнения

6.4.1

Воспользовавшись в качестве образца рис. 6.4, проиллюстрируйте работу процедуры HEAPSORT над входным массивом $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4.2

Докажите корректность процедуры HEAPSORT с помощью следующего инварианта цикла.

> В начале каждой итерации цикла **for** в строках 2–5 подмассив $A[1..i]$ представляет собой невозрастающую пирамиду, содержащую i наименьших элементов массива $A[1..n]$, а в подмассиве $A[i+1..n]$ содержатся $n-i$ наибольших элементов массива $A[1..n]$ в отсортированном состоянии.

6.4.3

Чему равно время работы процедуры HEAPSORT с массивом A длиной n, в котором элементы отсортированы и расположены в порядке возрастания? В порядке убывания?

6.4.4

Покажите, что время работы процедуры HEAPSORT в наихудшем случае составляет $\Omega(n \lg n)$.

6.4.5 ★

Покажите, что, когда все элементы различны, время работы процедуры HEAPSORT в наилучшем случае равно $\Omega(n \lg n)$.

6.5. Очереди с приоритетами

Пирамидальная сортировка — превосходный алгоритм, однако качественная реализация алгоритма быстрой сортировки, представленного в главе 7, на практике обычно превосходит по производительности пирамидальную сортировку. Тем не менее структура данных, использующаяся при пирамидальной сортировке, сама по себе имеет множество применений. В этом разделе представлено одно из наиболее популярных применений пирамид — в качестве эффективных очередей с приоритетами. Как и пирамиды, очереди с приоритетами бывают двух видов: невозрастающие и неубывающие. Мы рассмотрим процесс реализации невозрастающих очередей с приоритетами, которые основаны на невозрастающих пирамидах; в упр. 6.5.3 требуется написать процедуры для неубывающих очередей с приоритетами.

Очередь с приоритетами (priority queue) представляет собой структуру данных, предназначенную для обслуживания множества S, с каждым элементом которого связано определенное значение, называющееся ***ключом*** (key). В ***невозрастающей очереди с приоритетами*** поддерживаются следующие операции.

INSERT(S, x) вставляет элемент x в множество S, что эквивалентно операции $S = S \cup \{x\}$.

MAXIMUM(S) возвращает элемент множества S с наибольшим ключом.

EXTRACT-MAX(S) удаляет и возвращает элемент множества S с наибольшим ключом.

INCREASE-KEY(S, x, k) увеличивает значение ключа элемента x до нового значения k, которое предполагается не меньшим значения текущего ключа элемента x.

Среди прочих областей применения невозрастающих очередей — планирование заданий на совместно используемом компьютере. Очередь позволяет следить за заданиями, которые подлежат выполнению, и за их относительными приоритетами. Если задание прервано или завершило свою работу, планировщик выбирает

из очереди с помощью операции EXTRACT-MAX следующее задание с наибольшим приоритетом. В очередь в любое время можно добавить новое задание, воспользовавшись операцией INSERT.

Аналогично в ***неубывающей очереди с приоритетами*** поддерживаются операции INSERT, MINIMUM, EXTRACT-MIN и DECREASE-KEY. Очереди этого вида могут использоваться в моделировании систем, управляемых событиями. В роли элементов очереди в таком случае выступают моделируемые события, для каждого из которых сопоставляется время осуществления, играющее роль ключа. События должны моделироваться последовательно, согласно времени их наступления, поскольку процесс моделирования может вызвать генерацию других событий, которые нужно будет моделировать позже. Моделирующая программа выбирает очередное моделируемое событие с помощью операции EXTRACT-MIN. Когда инициируются новые события, они помещаются в очередь с помощью процедуры INSERT. В главах 23 и 24 нам предстоит познакомиться и с другими случаями применения неубывающих очередей с приоритетами, когда особо важной становится роль операции DECREASE-KEY.

Не удивительно, что приоритетную очередь можно реализовать с помощью пирамиды. В каждом отдельно взятом приложении, например в планировщике заданий, или при моделировании событий элементы очереди с приоритетами соответствуют объектам, с которыми работает это приложение. Часто возникает необходимость определить, какой из объектов приложения отвечает тому или иному элементу очереди, или наоборот. Если очередь с приоритетами реализуется с помощью пирамиды, то в каждом элементе пирамиды приходится хранить ***идентификатор*** (handle) соответствующего объекта приложения. То, каким будет конкретный вид этого идентификатора (таким, как указатель или целочисленный индекс), зависит от приложения. В каждом объекте приложения точно так же необходимо хранить идентификатор соответствующего элемента пирамиды. В данной книге таким идентификатором, как правило, будет индекс массива. Поскольку в ходе операций над пирамидой ее элементы изменяют свое расположение в массиве, при перемещении элемента пирамиды необходимо также обновлять значение индекса в соответствующем объекте приложения. Так как детали доступа к объектам приложения сильно зависят от самого приложения и его реализации, мы не станем останавливаться на этом вопросе. Ограничимся лишь замечанием, что на практике необходима организация надлежащей обработки идентификаторов.

Теперь рассмотрим, как реализовать операции в невозрастающей очереди с приоритетами. Процедура HEAP-MAXIMUM реализует операцию MAXIMUM за время $\Theta(1)$.

HEAP-MAXIMUM(A)
return $A[1]$

Процедура HEAP-EXTRACT-MAX реализует операцию EXTRACT-MAX. Она похожа на тело цикла **for** (строки 3–5) процедуры HEAPSORT.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "Очередь пуста"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Время работы процедуры HEAP-EXTRACT-MAX равно $O(\lg n)$, поскольку она выполняет только константное количество работы перед вызовом процедуры MAX-HEAPIFY, время работы которой — $O(\lg n)$.

Процедура HEAP-INCREASE-KEY реализует операцию INCREASE-KEY. Элемент очереди с приоритетами, ключ которого подлежит увеличению, идентифицируется в массиве с помощью индекса i. Сначала процедура обновляет ключ элемента $A[i]$. Поскольку это действие может нарушить свойство невозрастающей пирамиды, после этого процедура проходит путь от измененного узла к корню в поисках надлежащего места для нового ключа. Эта операция напоминает операцию, реализованную в цикле процедуры INSERTION-SORT из раздела 2.1 (строки 5–7). В процессе прохода выполняется сравнение текущего элемента с родительским. Если оказывается, что ключ текущего элемента превышает значение ключа родительского элемента, то происходит обмен ключами элементов и процедура продолжает свою работу на более высоком уровне. В противном случае процедура прекращает работу, поскольку ей удалось восстановить свойство невозрастающих пирамид. (Точная формулировка соответствующего инварианта цикла приведена в упр. 6.5.5.)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "Новый ключ меньше текущего"
3  A[i] = key
4  while i > 1 и A[PARENT(i)] < A[i]
5      Обменять A[i] и A[PARENT(i)]
6      i = PARENT(i)
```

На рис. 6.5 показан пример выполнения операции HEAP-INCREASE-KEY. Время работы процедуры HEAP-INCREASE-KEY над n-элементной пирамидой равно $O(\lg n)$, поскольку путь от обновляемого в строке 3 узла до корня имеет длину $O(\lg n)$.

Процедура MAX-HEAP-INSERT реализует операцию INSERT. В качестве параметра этой процедуре передается ключ нового элемента, вставляемого в невозрастающую пирамиду A. Сначала процедура добавляет в пирамиду новый лист и присваивает ему ключ со значением $-\infty$. Затем вызывается процедура HEAP-INCREASE-KEY, которая присваивает корректное значение ключу нового узла и помещает его в надлежащее место, чтобы не нарушалось свойство невозрастающей пирамиды.

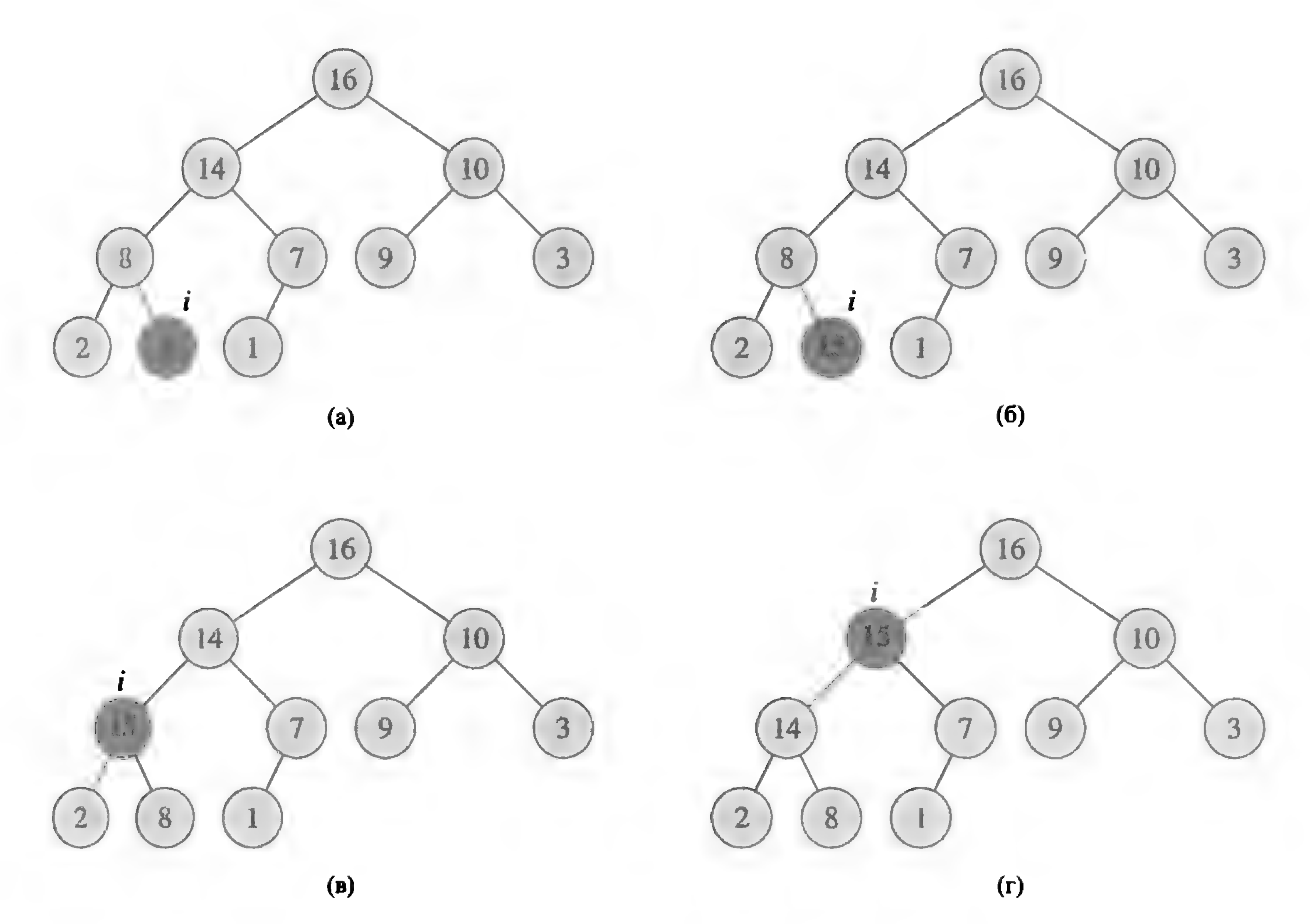


Рис. 6.5. Операция Heap-Increase-Key. **(а)** Невозрастающая пирамида с рис. 6.4, (а), с заштрихованным узлом с индексом i. **(б)** Ключ этого узла увеличивается до 15. **(в)** После одной итерации цикла **while** в строках 4–6 узел и его родитель обмениваются ключами, и индекс i перемещается в родительский узел. **(г)** Невозрастающая пирамида после еще одной итерации цикла **while**. В этот момент $A[\text{Parent}(i)] \geq A[i]$. Теперь свойство невозрастающей пирамиды выполняется, и процедура завершает работу.

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Время работы процедуры Max-Heap-Insert с n-элементной пирамидой составляет $O(\lg n)$.

Подводя итог, заметим, что время выполнения всех операций по обслуживанию очереди с приоритетами в пирамиде равно $O(\lg n)$.

Упражнения

6.5.1

Проиллюстрируйте работу процедуры Heap-Extract-Max над пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5.2

Проиллюстрируйте операцию Max-Heap-Insert$(A, 10)$ над пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7 Зак. 3726

6.5.3

Напишите псевдокоды процедур HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY и MIN-HEAP-INSERT, реализующих неубывающую очередь с приоритетами на базе неубывающей пирамиды.

6.5.4

Зачем нужна такая мера предосторожности, как присвоение ключу добавляемого в строке 2 процедуры MAX-HEAP-INSERT в пирамиду узла значения $-\infty$, если уже на следующем шаге значение этого ключа увеличивается до требуемой величины?

6.5.5

Докажите корректность процедуры HEAP-INCREASE-KEY с помощью следующего инварианта цикла.

> В начале каждой итерации цикла **while** в строках 4–6 $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ и $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, если эти узлы существуют, а подмассив $A[1\,..\,A.\mathit{heap\text{-}size}]$ удовлетворяет свойству невозрастающей пирамиды, за исключением, возможно, одного нарушения: $A[i]$ может быть больше, чем $A[\text{PARENT}(i)]$.

Можно считать, что в момент вызова процедуры HEAP-INCREASE-KEY подмассив $A[1\,..\,A.\mathit{heap\text{-}size}]$ удовлетворяет свойству невозрастающей пирамиды.

6.5.6

Каждая операция обмена в строке 5 процедуры HEAP-INCREASE-KEY обычно требует трех присваиваний. Покажите, как воспользоваться идеей внутреннего цикла процедуры INSERTION-SORT, чтобы вместо трех присваиваний обойтись только одним.

6.5.7

Покажите, как с помощью очереди с приоритетами реализовать очередь "первым вошел — первым вышел". Продемонстрируйте, как с помощью очереди с приоритетами реализовать стек. (Очереди и стеки определены в разделе 10.1.)

6.5.8

Процедура HEAP-DELETE(A, i) удаляет из пирамиды A узел i. Разработайте реализацию этой процедуры, которой требуется время $O(\lg n)$ для удаления узла из n-элементной невозрастающей пирамиды.

6.5.9

Разработайте алгоритм, объединяющий k отсортированных списков в один список за время $O(n \lg k)$, где n — общее количество элементов во всех входных списках. (*Указание:* для слияния списков воспользуйтесь неубывающей пирамидой.)

Задачи

6.1. Построение пирамиды вставками

Пирамиду можно построить с помощью многократного вызова процедуры MAX-HEAP-INSERT для вставки элементов в пирамиду. Рассмотрим следующий вариант процедуры BUILD-MAX-HEAP.

```
BUILD-MAX-HEAP'(A)
A.heap-size = 1
for i = 2 to A.length
    MAX-HEAP-INSERT(A, A[i])
```

а. Всегда ли процедуры BUILD-MAX-HEAP и BUILD-MAX-HEAP′ для одного и того же входного массива создают одну и ту же пирамиду? Докажите, что это так, или приведите контрпример.

б. Покажите, что в наихудшем случае для создания n-элементной пирамиды процедуре BUILD-MAX-HEAP′ потребуется время $\Theta(n \lg n)$.

6.2. Анализ d-арных пирамид

***d*-арные пирамиды** подобны бинарным, но отличаются тем, что все внутренние узлы (с одним возможным исключением) имеют вместо двух d дочерних узлов.

а. Как бы вы представили d-арную пирамиду в виде массива?

б. Как выражается высота d-арной n-элементной пирамиды через n и d?

в. Разработайте эффективную реализацию процедуры EXTRACT-MAX, предназначенную для работы с d-арной невозрастающей пирамидой. Проанализируйте время работы этой процедуры и выразите его через d и n.

г. Разработайте эффективную реализацию процедуры INSERT, предназначенную для работы с d-арной невозрастающей пирамидой. Проанализируйте время работы этой процедуры и выразите его через d и n.

д. Разработайте эффективную реализацию процедуры INCREASE-KEY(A, i, k), которая при $k < A[i]$ сообщает об ошибке, а в противном случае выполняет присваивание $A[i] = k$ и соответствующим образом обновляет структуру d-арной невозрастающей пирамиды. Проанализируйте время работы этой процедуры и выразите его через d и n.

6.3. Таблицы Юнга

Таблица Юнга (Young tableau) $m \times n$ представляет собой матрицу размером $m \times n$, элементы которой в каждой строке отсортированы слева направо, а в каж-

дом столбце — сверху вниз. Некоторые элементы таблицы Юнга могут быть равны ∞, что трактуется как отсутствие элемента. Таким образом, таблицу Юнга можно использовать для хранения $r \leq mn$ конечных чисел.

а. Начертите таблицу Юнга 4×4, в которой содержатся элементы $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

б. Докажите, что таблица Юнга Y размером $m \times n$ пуста, если $Y[1, 1] = \infty$. Докажите, что таблица Y полностью заполнена (т.е. содержит mn элементов), если $Y[m, n] < \infty$.

в. Разработайте алгоритм, реализующий процедуру EXTRACT-MIN для непустой таблицы Юнга $m \times n$ за время $O(m + n)$. В алгоритме следует использовать рекурсивную подпрограмму, которая решает задачу размером $m \times n$ путем рекурсивного сведéния к задачам $(m - 1) \times n$ или $m \times (n - 1)$. (*Указание:* вспомните о процедуре MAX-HEAPIFY.) Обозначим максимальное время обработки произвольной таблицы Юнга $m \times n$ с помощью процедуры EXTRACT-MIN как $T(p)$, где $p = m + n$. Запишите и решите рекуррентное соотношение, которое дает для $T(p)$ границу $O(m + n)$.

г. Покажите, как вставить новый элемент в незаполненную таблицу Юнга размером $m \times n$ за время $O(m + n)$.

д. Покажите, как с помощью таблицы Юнга $n \times n$ выполнить сортировку n^2 чисел за время $O(n^3)$, не используя при этом никаких других подпрограмм сортировки.

е. Разработайте алгоритм, позволяющий за время $O(m + n)$ определить, содержится ли в таблице Юнга размером $m \times n$ заданное число.

Заключительные замечания

Алгоритм пирамидальной сортировки был разработан Вильямсом (Williams) [355], который также описал, каким образом с помощью пирамиды можно реализовать очередь с приоритетами. Процедура BUILD-MAX-HEAP предложена Флойдом (Floyd) [105].

В главах 16, 23 и 24 неубывающие пирамиды будут использованы для реализации неубывающих очередей с приоритетами. В главе 19 будет представлена реализация с улучшенными временны́ми границами, а в главе 20 — усовершенствованная реализация для случая, когда ключи выбираются из ограниченного множества неотрицательных целых чисел.

Для случая, когда данные представляют собой b-битовые целые числа, а память компьютера состоит из адресуемых b-битовых слов, Фредман (Fredman) и Уиллард (Willard) [114] показали, как реализовать процедуру MINIMUM со временем работы $O(1)$ и процедуры INSERT и EXTRACT-MIN со временем работы

$O(\sqrt{\lg n})$. Торуп (Thorup) [335] улучшил границу $O(\sqrt{\lg n})$ до $O(\lg \lg n)$. При этом используемая память не ограничена величиной n, однако такого линейного ограничения используемой памяти можно достичь с помощью рандомизированного хеширования.

Важный частный случай очередей с приоритетами имеет место, когда последовательность операций EXTRACT-MIN является ***монотонной***, т.е. возвращаемые последовательными операциями EXTRACT-MIN значения образуют монотонно неубывающую последовательность. Такая ситуация встречается во многих важных приложениях, например в алгоритме поиска кратчайшего пути Дейкстры (Dijkstra), который рассматривается в главе 24, или при моделировании дискретных событий. Для алгоритма Дейкстры особенно важна эффективность реализации операции DECREASE-KEY. Для частного случая монотонности для целочисленных данных из диапазона $1, 2, \ldots, C$ Ахуйя (Ahuja), Мельхорн (Mehlhorn), Орлин (Orlin) и Таржан (Tarjan) [8] описали, как с помощью структуры данных под названием "позиционная пирамида" (radix heap) реализовать операции EXTRACT-MIN и INSERT с амортизированным временем работы $O(\lg C)$ (более подробные сведения на эту тему можно найти в главе 17) и операцию DECREASE-KEY со временем работы $O(1)$. Граница $O(\lg C)$ может быть улучшена до $O(\sqrt{\lg C})$ путем совместного использования пирамид Фибоначчи (см. главу 19) и позиционных пирамид. Дальнейшее улучшение этой границы до $O(\lg^{1/3+\varepsilon} C)$ было осуществлено Черкасски (Cherkassky), Гольдбергом (Goldberg) и Сильверстайном (Silverstein) [64], которые объединили многоуровневую группирующую структуру (multi-level bucketing structure) Денардо (Denardo) и Фокса (Fox) [84] с уже упоминавшейся пирамидой Торупа. Раману (Raman) [289] удалось еще больше улучшить эти результаты и получить границу, которая равна $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$ для произвольной фиксированной величины $\epsilon > 0$.

Глава 7. Быстрая сортировка

Алгоритм быстрой сортировки имеет в наихудшем случае для входного массива из n элементов время работы, равное $\Theta(n^2)$. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм на практике зачастую оказывается оптимальным благодаря тому, что в среднем время его работы намного лучше: $\Theta(n \lg n)$. Кроме того, постоянный множитель, скрытый в выражении $\Theta(n \lg n)$, достаточно мал по величине. Алгоритм обладает также тем преимуществом, что сортировка в нем выполняется на месте, без использования дополнительной памяти (см. с. 39), поэтому он хорошо работает даже в средах с виртуальной памятью.

В разделе 7.1 описаны сам алгоритм и важная подпрограмма, использующаяся в нем для разбиения массива. Поскольку поведение алгоритма быстрой сортировки достаточно сложное, мы начнем с нестрогого, интуитивного обсуждения производительности этого алгоритма в разделе 7.2, а строгий анализ отложим до конца данной главы. В разделе 7.3 представлена версия быстрой сортировки, в которой используется случайная выборка. У этого алгоритма хорошее ожидаемое время работы, при этом никакие конкретные входные данные не могут ухудшить его производительность до уровня наихудшего случая. Этот рандомизированный алгоритм анализируется в разделе 7.4, где показано, что время его работы в наихудшем случае равно $\Theta(n^2)$, а среднее время работы в предположении, что все элементы различны, составляет $O(n \lg n)$.

7.1. Описание быстрой сортировки

Быстрая сортировка, подобно сортировке слиянием, применяет парадигму "разделяй и властвуй", представленную в разделе 2.3.1. Ниже описан процесс сортировки подмассива $A[p\,..\,r]$, состоящий, как и все алгоритмы с использованием декомпозиции, из трех этапов.

Разделение. Массив $A[p\,..\,r]$ разбивается на два (возможно, пустых) подмассива $A[p\,..\,q-1]$ и $A[q+1\,..\,r]$, таких, что каждый элемент $A[p\,..\,q-1]$ меньше или равен $A[q]$, который, в свою очередь, не превышает любой элемент подмассива $A[q+1\,..\,r]$. Индекс q вычисляется в ходе процедуры разбиения.

Властвование. Подмассивы $A[p\,..\,q-1]$ и $A[q+1\,..\,r]$ сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.

Комбинирование. Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив $A[p\,..\,r]$ оказывается отсортированным.

Быстрая сортировка реализуется следующей процедурой.

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Для сортировки всего массива A начальный вызов процедуры должен иметь вид QUICKSORT($A, 1, A.\mathit{length}$).

Разбиение массива

Ключевой частью рассматриваемого алгоритма сортировки является процедура PARTITION, изменяющая порядок элементов подмассива $A[p\,..\,r]$ без привлечения дополнительной памяти.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          Обменять A[i] и A[j]
7  Обменять A[i + 1] и A[r]
8  return i + 1
```

На рис. 7.1 показано, как процедура PARTITION работает с 8-элементным массивом. Эта процедура всегда выбирает элемент $x = A[r]$ в качестве ***опорного*** (pivot). Разбиение подмассива $A[p\,..\,r]$ будет выполняться относительно этого элемента. В начале выполнения процедуры массив разделяется на четыре области (они могут быть пустыми). В начале каждой итерации цикла **for** в строках 3–6 каждая область удовлетворяет определенным свойствам, показанным на рис. 7.2. Эти свойства можно сформулировать в виде инварианта цикла.

В начале каждой итерации цикла в строках 3–6 для любого индекса k массива справедливо следующее:

1. если $p \le k \le i$, то $A[k] \le x$;
2. если $i + 1 \le k \le j - 1$, то $A[k] > x$;
3. если $k = r$, то $A[k] = x$.

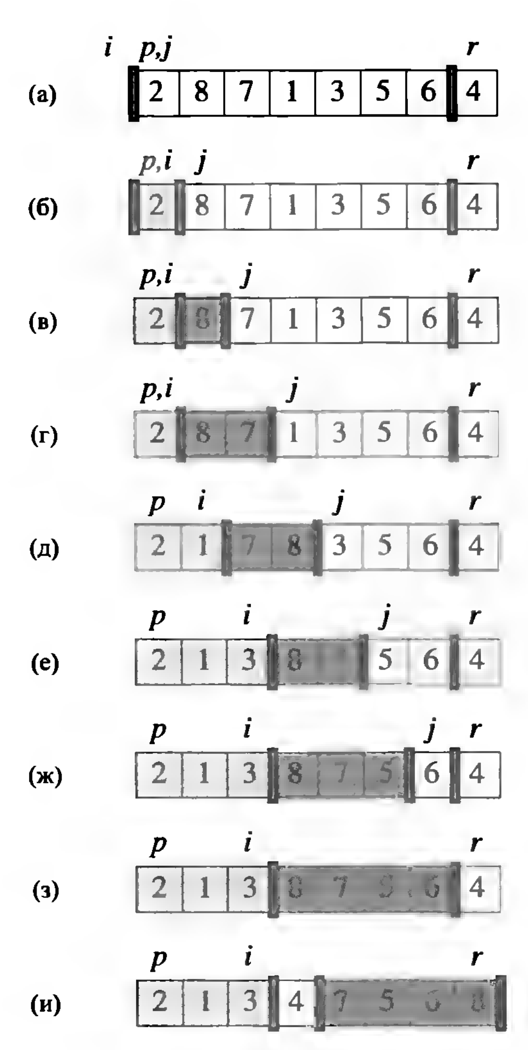


Рис. 7.1. Пример действия процедуры PARTITION на массив. Элемент массива $A[r]$ становится опорным элементом x. Светло-серым цветом обозначены элементы массива, которые попали в первую часть разбиения; их значения не превышают x. Элементы темно-серого цвета образуют вторую часть массива; их величина больше x. Незакрашенные элементы — это элементы, которые пока что не попали ни в одну из первых двух частей; последний незакрашенный элемент является опорным элементом x. **(а)** Начальное состояние массива и значения переменных. Ни один элемент не помещен ни в одну из первых двух частей. **(б)** Элемент со значением 2 "переставлен сам с собой" и помещен в часть с меньшими значениями. **(в)** и **(г)** Элементы со значениями 8 и 7 добавлены в часть с бóльшими значениями. **(д)** Элементы 1 и 8 поменялись местами, в результате чего количество элементов в первой части возросло. **(е)** Обмен местами элементов 3 и 7, в результате чего количество элементов в первой части возрастает. **(ж)** и **(з)** Вторая часть увеличивается за счет включения в нее элементов 5 и 6, после чего цикл завершается. **(и)** В строках 7 и 8 опорный элемент меняется местами с тем, который находится между двумя областями.

Индексы между j и $r - 1$ не подпадают ни под один из трех перечисленных случаев, и значения соответствующих им элементов не имеют определенной связи с опорным элементом x.

Нам необходимо показать, что сформулированный выше инвариант цикла справедлив перед первой итерацией, что каждая итерация цикла сохраняет этот инвариант и что он позволяет продемонстрировать корректность алгоритма по завершении цикла.

Инициализация. Перед первой итерацией цикла $i = p - 1$ и $j = p$. Между элементами с индексами p и i нет никаких элементов, как нет их и между

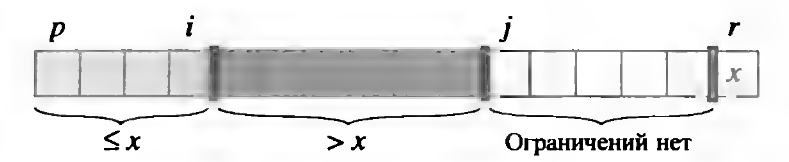


Рис. 7.2. Четыре области, поддерживаемые процедурой PARTITION в подмассиве $A[p..r]$. Все элементы подмассива $A[p..i]$ меньше либо равны x, все элементы подмассива $A[i+1..j-1]$ больше x, а $A[r] = x$. Подмассив $A[j..r-1]$ может иметь любые значения.

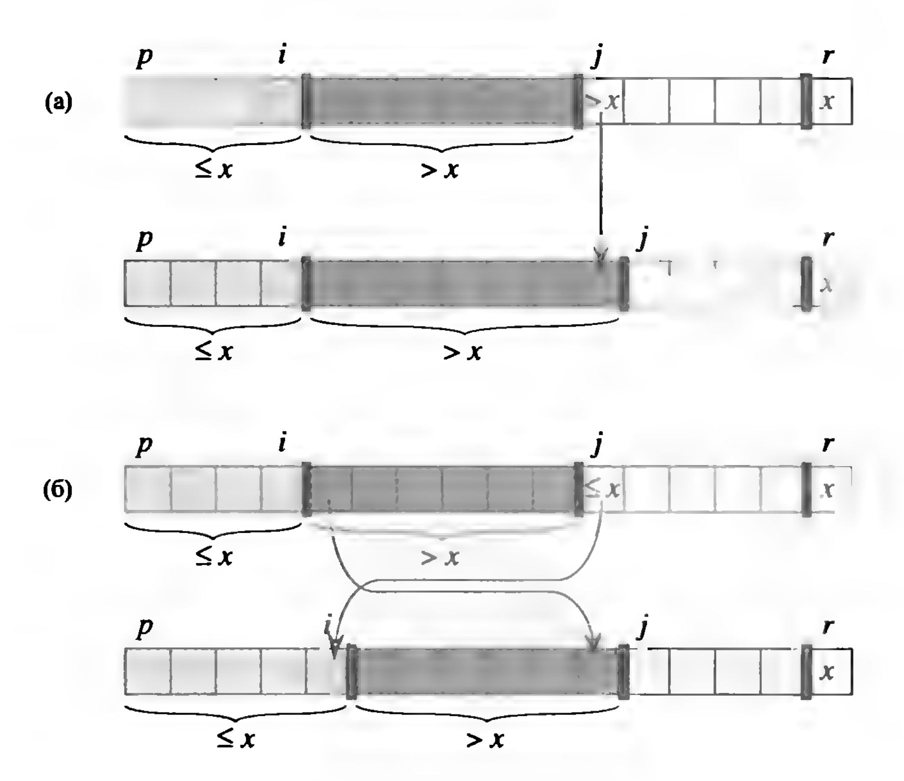


Рис. 7.3. Два варианта итерации в процедуре PARTITION. **(а)** Если $A[j] > x$, единственным действием является увеличение j, что сохраняет инвариант цикла. **(б)** Если $A[j] \le x$, индекс i увеличивается, $A[i]$ и $A[j]$ меняются местами, после чего увеличивается j. Инвариант цикла сохраняется и в этом случае.

элементами с индексами $i + 1$ и $j - 1$, поэтому первые два условия инварианта цикла тривиально выполняются. Присваивание в строке 1 удовлетворяет третьему условию.

Сохранение. Как видно из рис. 7.3, нужно рассмотреть два случая, выбор одного из которых определяется проверкой в строке 4. На рис. 7.3, (а) показано, что происходит, если $A[j] > x$; единственное действие, которое выполняется в этом случае в цикле, — это увеличение на единицу значения j. При увеличении значения j условие 2 выполняется для элемента $A[j-1]$, а все остальные элементы остаются неизменными. На рис. 7.3, (б) показано, что происходит, если $A[j] \le x$; в этом случае увеличивается значение i, элементы $A[i]$ и $A[j]$ меняются местами, после чего на единицу увеличивается значение j. В результате перестановки получаем $A[i] \le x$, и условие 1 выполняется. Аналогично получаем $A[j-1] > x$, поскольку элемент, который был переставлен в позицию элемента $A[j-1]$, согласно инварианту цикла больше x.

Завершение. По завершении работы алгоритма $j = r$. Поэтому каждый элемент массива является членом одного из трех множеств, описанных в инварианте цикла. Таким образом, все элементы массива разбиты на три множества: величина которых не превышает x, превышающие x и одноэлементное множество, состоящее из элемента x.

В последних двух строках процедуры PARTITION опорный элемент перемещается на свое место в средину массива с помощью его перестановки с крайним левым элементом, превышающим величину x. После этого процедура возвращает новый индекс опорного элемента. Выход процедуры PARTITION удовлетворяет требованиям, наложенным шагом разделения. Фактически он удовлетворяет немного более строгому условию: после строки 2 процедуры QUICKSORT элемент $A[q]$ строго меньше любого элемента $A[q+1\,..\,r]$.

Время работы процедуры PARTITION над подмассивом $A[p\,..\,r]$ равно $\Theta(n)$, где $n = r - p + 1$ (см. упр. 7.1.3).

Упражнения

7.1.1

Воспользовавшись рис. 7.1 в качестве образца, проиллюстрируйте работу процедуры PARTITION с массивом $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1.2

Какое значение q возвращает процедура PARTITION, если все элементы массива $A[p\,..\,r]$ одинаковы? Модифицируйте эту процедуру так, чтобы в случае, когда все элементы массива $A[p\,..\,r]$ имеют одно и то же значение, q определялось следующим образом: $q = \lfloor (p+r)/2 \rfloor$.

7.1.3

Приведите краткое обоснование утверждения, что время обработки процедурой PARTITION подмассива, состоящего из n элементов, равно $\Theta(n)$.

7.1.4

Как бы вы изменили процедуру QUICKSORT для сортировки в невозрастающем порядке?

7.2. Производительность быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от степени сбалансированности, которой характеризуется разбиение. Сбалансированность, в свою очередь, зависит от того, какой элемент выбран в качестве опорного. Если разбиение сбалансированное, асимптотически алгоритм работает так же быстро, как и сортировка слиянием. В противном случае асимптотическое поведение этого алгоритма столь же медленное, как и у сортировки вставкой. В данном разделе

будет проведено неформальное исследование поведения быстрой сортировки при условии сбалансированного и несбалансированного разбиений.

Разбиение в наихудшем случае

Наихудшее поведение алгоритма быстрой сортировки имеет место в случае, когда подпрограмма, выполняющая разбиение, порождает одну подзадачу с $n-1$ элементами, а вторую — пустую. (Это будет доказано в разделе 7.4.1.) Предположим, что такое несбалансированное разбиение возникает при каждом рекурсивном вызове. Для выполнения разбиения требуется время $\Theta(n)$. Поскольку рекурсивный вызов процедуры разбиения, на вход которой подается массив размером 0, приводит к немедленному возврату из этой процедуры без выполнения каких-либо операций, $T(0) = \Theta(1)$. Таким образом, рекуррентное соотношение, описывающее время работы процедуры в указанном случае, записывается следующим образом:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned}$$

Интуитивно понятно, что при суммировании промежутков времени, затрачиваемых на каждый уровень рекурсии, получается арифметическая прогрессия (уравнение (A.2)), что приводит к результату $\Theta(n^2)$. В самом деле, с помощью метода подстановок легко доказать, что решением рекуррентного соотношения $T(n) = T(n-1) + \Theta(n)$ является $T(n) = \Theta(n^2)$ (см. упр. 7.2.1).

Таким образом, если на каждом уровне рекурсии алгоритма разбиение максимально несбалансированное, то время работы алгоритма равно $\Theta(n^2)$. Следовательно, производительность быстрой сортировки в наихудшем случае не превышает производительности сортировки вставкой. Более того, это же время требуется алгоритму быстрой сортировки для обработки уже полностью отсортированного массива (распространенная ситуация, в которой время работы алгоритма сортировки вставкой равно $O(n)$).

Разбиение в наилучшем случае

В наиболее благоприятном случае процедура PARTITION приводит к двум подзадачам, размер каждой из которых не превышает $n/2$, поскольку размер одной из них равен $\lfloor n/2 \rfloor$, а второй — $\lceil n/2 \rceil - 1$. В такой ситуации быстрая сортировка работает намного производительнее, и время ее работы описывается следующим рекуррентным соотношением:

$$T(n) = 2T(n/2) + \Theta(n) \ ,$$

где мы не обращаем внимания на неточность, связанную с игнорированием функций "пол" и "потолок", и вычитанием 1. В соответствии со случаем 2 основной теоремы (теорема 4.1) это рекуррентное соотношение имеет решение $T(n) = \Theta(n \lg n)$. При сбалансированности двух частей разбиения на каждом уровне рекурсии мы получаем асимптотически более быстрый алгоритм.

Сбалансированное разбиение

Как станет ясно из анализа, проведенного в разделе 7.4, в асимптотическом пределе поведение алгоритма быстрой сортировки в среднем случае намного ближе к его поведению в наилучшем случае, чем к поведению в наихудшем случае. Чтобы стало ясно, почему это так, нужно понять, как баланс разбиения отражается на рекуррентном соотношении, описывающем время работы алгоритма.

Предположим, например, что разбиение всегда происходит в соотношении один к девяти, что, на первый взгляд, весьма далеко от сбалансированности. В этом случае для времени работы алгоритма быстрой сортировки мы получим рекуррентное соотношение

$$T(n) = T(9n/10) + T(n/10) + cn ,$$

в которое явным образом входит константа c, скрытая в члене $\Theta(n)$. На рис. 7.4 показано дерево рекурсии, отвечающее этому рекуррентному соотношению. Обратите внимание, что каждый уровень этого дерева имеет стоимость cn, и так до тех пор, пока не будет достигнуто граничное условие на глубине $\log_{10} n = \Theta(\lg n)$; после этого стоимость более глубоких уровней не превышает величину cn. Рекурсия прекращается на глубине $\log_{10/9} n = \Theta(\lg n)$; таким образом, общая стоимость быстрой сортировки равна $O(n \lg n)$. Итак, если на каждом уровне рекурсии разбиение производится в соотношении один к девяти (что интуитивно воспринимается как сильный дисбаланс), время работы алгоритма быстрой сортировки равно $O(n \lg n)$ — асимптотически такое же, как и при сбалансированном разбиении. Фактически любое разбиение, характеризующееся конечной *константой* пропорциональности, приводит к образованию дерева рекурсии высотой $\Theta(\lg n)$ со стоимостью каждого уровня, равной $O(n)$. Следовательно, при любой постоянной пропорции разбиения полное время работы быстрой сортировки составляет $O(n \lg n)$.

Интуитивные рассуждения для среднего случая

Чтобы получить ясное представление о рандомизированном поведении алгоритма быстрой сортировки, необходимо сделать предположение о том, как часто ожидается появление тех или иных входных данных. Поведение быстрой сортировки зависит от относительного расположения значений во входном массиве, а не от их конкретных величин. Как и при вероятностном анализе задачи о найме в разделе 5.2, пока что предположим, что все перестановки входных чисел равновероятны.

Если алгоритм быстрой сортировки работает со случайным образом выбранным входным массивом, то маловероятно, чтобы разбиение на каждом уровне происходило в одном и том же соотношении, как это было в ходе проведенного выше неформального анализа. Ожидается, что одни разбиения будут хорошо сбалансированы, в то время как другие окажутся сбалансированными плохо. Например, в упр. 7.2.6 нужно показать, что соотношение размеров подзадач на выходе процедуры PARTITION примерно в 80% случаев сбалансировано лучше, чем девять к одному, и примерно в 20% случаев — хуже.

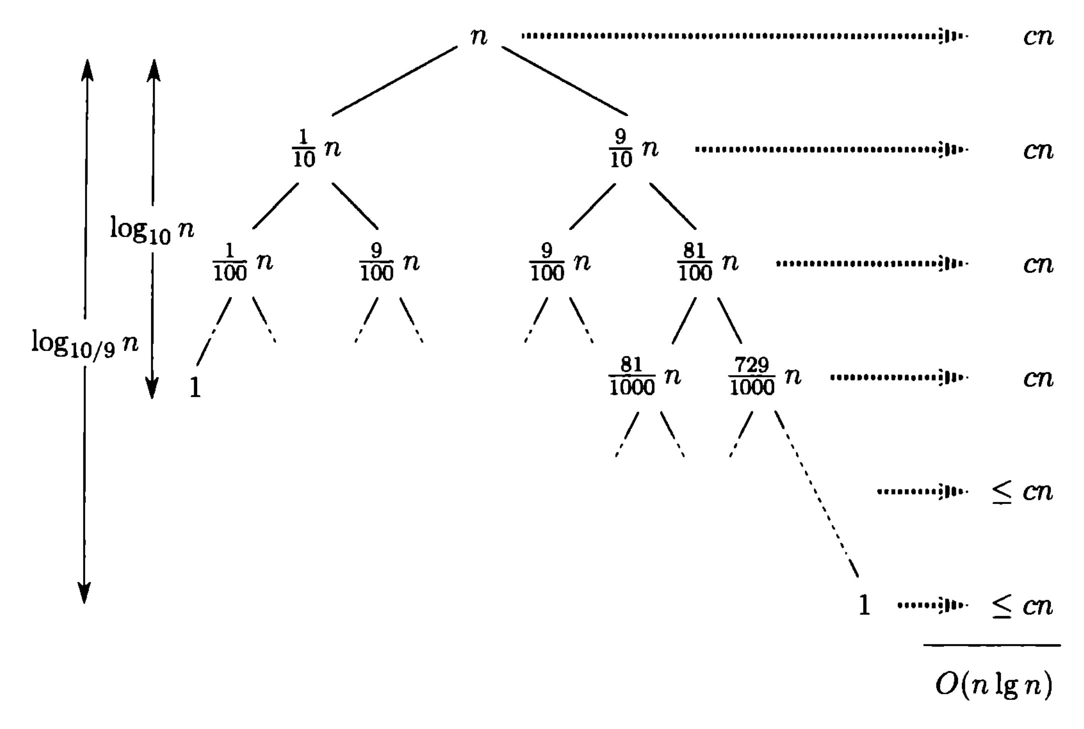


Рис. 7.4. Дерево рекурсии для алгоритма QUICKSORT, в котором процедура PARTITION всегда выполняет разбиение в соотношении девять к одному, дающее время работы $O(n \lg n)$. Узлы показывают размеры подзадач; стоимости уровней показаны справа. Стоимость уровня неявно включает константу c в члене $\Theta(n)$.

В среднем случае процедура PARTITION производит как "хорошие", так и "плохие" деления. В дереве рекурсии, соответствующем среднему случаю, хорошие и плохие разбиения равномерно распределены по всему дереву. Для упрощения интуитивных рассуждений предположим, что уровни дерева с плохими и хорошими разбиениями чередуются, и что хорошие разбиения получаются такими, как в наилучшем случае, а плохие — такими, как в наихудшем. На рис. 7.5, (а) показаны разбиения на двух соседних уровнях такого дерева рекурсии. В корне дерева стоимость разбиения равна n, а размеры полученных подмассивов равны $n - 1$ и 0, что соответствует наихудшему случаю. На следующем уровне подмассив размером $n - 1$ делится оптимальным образом на подмассивы размерами $(n-1)/2-1$ и $(n-1)/2$. Будем считать, что для подмассива размером 0 стоимость равна 1.

Комбинация плохого и хорошего разбиений приводит к образованию трех подмассивов с размерами 0, $(n - 1)/2 - 1$ и $(n - 1)/2$ и суммарной стоимостью $\Theta(n) + \Theta(n - 1) = \Theta(n)$. Эта ситуация, определенно, не хуже показанной на рис. 7.5, (б), на котором представлен один уровень разбиения со стоимостью $\Theta(n)$, создающий два подмассива размерами $(n - 1)/2$. Однако в этом случае разбиение получается сбалансированным! Интуитивно понятно, что время $\Theta(n-1)$, которое требуется для плохого разбиения, поглощается временем $\Theta(n)$, которое требуется для хорошего разбиения, а полученное в результате разбиение оказывается хорошим. Таким образом, время работы алгоритма быстрой сортировки, при которой на последовательных уровнях чередуются плохие и хорошие разбиения, ведет себя так же, как время работы быстрой сортировки для одних лишь хороших разбиений. Оно также равно $O(n \lg n)$, просто константа, скрытая в O-обозначении,

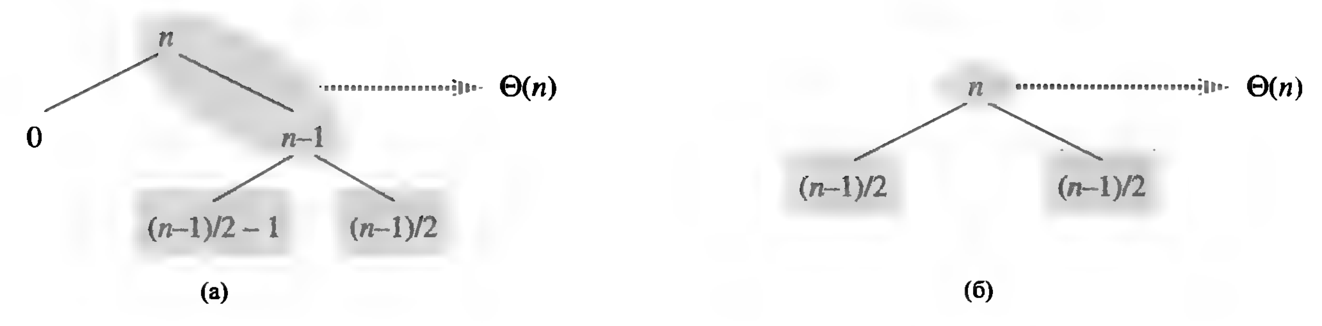


Рис. 7.5. **(а)** Два уровня дерева рекурсии для быстрой сортировки. Стоимость разбиения в корне равна n и генерирует плохое разбиение, т.е. подмассивы размерами 0 и $n-1$. Разбиение подмассива размером $n-1$ имеет стоимость $n-1$ и генерирует хорошее разбиение — подмассивы размерами $(n-1)/2-1$ и $(n-1)/2$. **(б)** Хорошо сбалансированный уровень дерева рекурсии. В обеих частях стоимость разбиения для задач в заштрихованных эллипсах составляет $\Theta(n)$. Задачи, которые остается решить в случае (а), и показанные в заштрихованных прямоугольниках, оказываются не больше подзадач, которые следует решить в случае (б).

в этом случае несколько больше. Строгий анализ ожидаемого времени работы рандомизированной версии быстрой сортировки содержится в разделе 7.4.2.

Упражнения

7.2.1

С помощью метода подстановки докажите, что решение рекуррентного соотношения $T(n) = T(n-1) + \Theta(n)$ имеет вид $T(n) = \Theta(n^2)$, как утверждалось в начале раздела 7.2.

7.2.2

Чему равно время работы процедуры QUICKSORT в случае, когда все элементы массива A одинаковы по величине?

7.2.3

Покажите, что если все элементы массива A различаются по величине и расположены в убывающем порядке, то время работы процедуры QUICKSORT равно $\Theta(n^2)$.

7.2.4

Сведения о банковских операциях часто записываются в хронологическом порядке, но многие предпочитают получать отчеты о состоянии своих банковских счетов в порядке нумерации чеков. Чеки чаще всего выписываются в порядке возрастания их номеров, а торговцы обычно обналичивают их с небольшой задержкой. Таким образом, задача преобразования упорядочения по времени операций в упорядочение по номерам чеков — это задача сортировки почти упорядоченных массивов. Обоснуйте утверждение, что при решении этой задачи процедура INSERTION-SORT обычно превосходит по производительности процедуру QUICKSORT.

7.2.5

Предположим, что в ходе быстрой сортировки на каждом уровне происходит разбиение в пропорции $1-\alpha$ к α, где $0 < \alpha \leq 1/2$ — константа. Покажите, что минимальная глубина, на которой расположен лист дерева рекурсии, приблизительно равна $-\lg n/\lg \alpha$, а максимальная глубина — приблизительно $-\lg n/\lg(1-\alpha)$. (Об округлении до целочисленных значений можно не беспокоиться.)

7.2.6 ★

Докажите, что для любой константы $0 < \alpha \leq 1/2$ вероятность того, что процедура Partition поделит случайно выбранный входной массив в более сбалансированной пропорции, чем $1-\alpha$ к α, приблизительно равна $1 - 2\alpha$.

7.3. Рандомизированная быстрая сортировка

Исследуя поведение алгоритма быстрой сортировки в среднем случае, мы делали предположение, что все перестановки входных чисел встречаются с равной вероятностью. Однако на практике это далеко не всегда так (см. упр. 7.2.4). Как мы знаем из раздела 5.3, можно добавить в алгоритм рандомизацию, чтобы получить хорошую ожидаемую производительность для всех входных данных. Многие считают такую рандомизированную версию алгоритма быстрой сортировки оптимальным выбором для обработки достаточно больших массивов.

В разделе 5.3 рандомизация алгоритма проводилась путем явной перестановки его входных элементов. В алгоритме быстрой сортировки можно было бы поступить точно так же, однако анализ упростится, если применить другой метод рандомизации, получивший название ***случайной выборки*** (random sampling). Вместо того чтобы в качестве опорного элемента всегда использовать $A[r]$, такой элемент будет выбираться в массиве $A[p \,..\, r]$ случайным образом. Подобная модификация, при которой опорный элемент выбирается случайным образом среди элементов с индексами от p до r, обеспечивает любому из $r - p + 1$ элементов подмассива равную вероятность оказаться опорным. Благодаря случайному выбору опорного элемента можно ожидать, что разбиение входного массива в среднем окажется довольно хорошо сбалансированным.

Изменения, которые нужно внести в процедуры Partition и Quicksort, незначительны. В новой версии процедуры разбиения непосредственно перед разбиением достаточно реализовать обмен.

```
Randomized-Partition(A, p, r)
  i = Random(p, r)
  Обменять A[r] и A[i]
  return Partition(A, p, r)
```

В новой процедуре быстрой сортировки вместо процедуры Partition вызывается процедура Randomized-Partition.

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Мы проанализируем этот алгоритм в следующем разделе.

Упражнения

7.3.1

Почему мы анализируем ожидаемое время работы рандомизированного алгоритма, а не его время работы в наихудшем случае?

7.3.2

Сколько раз в ходе выполнения процедуры RANDOMIZED-QUICKSORT в наихудшем случае вызывается генератор случайных чисел RANDOM? В наилучшем случае? Выразите свой ответ через Θ-обозначения.

7.4. Анализ быстрой сортировки

В разделе 7.2 были приведены некоторые интуитивные рассуждения по поводу поведения алгоритма быстрой сортировки в наихудшем случае и обосновывалось, почему следует ожидать достаточно высокой производительности его работы. В данном разделе проведен более строгий анализ поведения этого алгоритма. Начнем этот анализ с наихудшего случая. Подобный анализ применим как к процедуре QUICKSORT, так и к процедуре RANDOMIZED-QUICKSORT. Завершается раздел анализом ожидаемого времени работы процедуры RANDOMIZED-QUICKSORT.

7.4.1. Анализ наихудшего случая

В разделе 7.2 было показано, что при самом неудачном разбиении на каждом уровне рекурсии время работы алгоритма быстрой сортировки равно $\Theta(n^2)$. Интуитивно понятно, что это наихудшее время работы рассматриваемого алгоритма. Докажем это утверждение.

С помощью метода подстановки (см. раздел 4.3) можно показать, что время работы алгоритма быстрой сортировки равно $O(n^2)$. Пусть $T(n)$ — наихудшее время обработки процедурой QUICKSORT входных данных размером n. Тогда мы получаем следующее рекуррентное соотношение:

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) , \tag{7.1}$$

где параметр q изменяется в интервале от 0 до $n - 1$, поскольку на выходе процедуры PARTITION мы получаем две подзадачи, общий размер которых равен $n - 1$. Мы предполагаем, что $T(n) \leq cn^2$ для некоторой константы c. Подставив это неравенство в рекуррентное соотношение (7.1), получим

$$T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n)$$
$$= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \ .$$

Выражение $q^2 + (n - q - 1)^2$ достигает максимума на обоих концах интервала $0 \leq q \leq n - 1$, что подтверждается тем, что вторая производная от него по q положительна (см. упр. 7.4.3). Это наблюдение позволяет нам сделать оценку $\max_{0 \leq q \leq n-1}(q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$. Продолжая работу с границей для $T(n)$, получаем

$$T(n) \leq cn^2 - c(2n - 1) + \Theta(n)$$
$$\leq cn^2 \ ,$$

поскольку константу c можно выбрать настолько большой, чтобы слагаемое $c(2n - 1)$ доминировало над слагаемым $\Theta(n)$. Таким образом, $T(n) = O(n^2)$. В разделе 7.2 мы встречались с частным случаем быстрой сортировки, при котором требовалось время $\Omega(n^2)$, — это случай несбалансированного разбиения. В упр. 7.4.1 нужно показать, что рекуррентное соотношение (7.1) имеет решение $T(n) = \Omega(n^2)$. Таким образом, время работы алгоритма быстрой сортировки (в наихудшем случае) равно $\Theta(n^2)$.

7.4.2. Ожидаемое время работы

Мы уже приводили интуитивный аргумент в пользу того, что ожидаемое время работы процедуры RANDOMIZED-QUICKSORT равно $O(n \lg n)$: если на каждом уровне рекурсии в разделении, производимом процедурой RANDOMIZED_PARTITION, в одну часть массива помещается произвольная фиксированная доля элементов, то высота дерева рекурсии равна $\Theta(\lg n)$, а время работы каждого его уровня — $O(n)$. Даже после добавления некоторого количества новых уровней с наименее сбалансированным разбиением время работы останется равным $O(n \lg n)$. Можно провести точный анализ математического ожидания времени работы процедуры RANDOMIZED-QUICKSORT. Для этого сначала нужно понять, как работает процедура разбиения, а затем получить для математического ожидания времени работы оценку $O(n \lg n)$. Эта верхняя граница математического ожидания времени работы в сочетании с полученной в разделе 7.2 оценкой для наилучшего случая, равной $\Theta(n \lg n)$, позволяют сделать вывод о том, что математическое ожидание времени работы равно $\Theta(n \lg n)$. Мы предполагаем, что значения всех сортируемых элементов различны.

Время работы и сравнения

Процедуры QUICKSORT и RANDOMIZED-QUICKSORT отличаются только выбором опорного элемента; во всех прочих аспектах они идентичны. Таким образом, анализ процедуры RANDOMIZED-QUICKSORT можно провести, рассматривая процедуры QUICKSORT и PARTITION, но в предположении, что опорные элементы выбираются из передаваемого процедуре RANDOMIZED-PARTITION подмассива случайным образом.

Время работы процедуры QUICKSORT определяется преимущественно временем работы, затраченным на выполнение процедуры PARTITION. При каждом выполнении последней происходит выбор опорного элемента, который впоследствии не принимает участия ни в одном рекурсивном вызове процедур QUICKSORT и PARTITION. Таким образом, на протяжении всего времени выполнения алгоритма быстрой сортировки процедура PARTITION вызывается не более n раз. Один вызов процедуры PARTITION выполняется в течение времени $O(1)$, к которому нужно прибавить время, пропорциональное количеству итераций цикла **for** в строках 3–6. В каждой итерации цикла **for** в строке 4 опорный элемент сравнивается с другими элементами массива A. Поэтому, если известно общее количество выполнений строки 4, можно оценить полное время, которое затрачивается на выполнение цикла **for** в процессе работы процедуры QUICKSORT.

Лемма 7.1

Пусть X является количеством сравнений, выполняемых в строке 4 процедуры PARTITION за время работы процедуры QUICKSORT над n-элементным массивом. Тогда время работы процедуры QUICKSORT составляет $O(n + X)$.

Доказательство. Как следует из приведенных выше рассуждений, процедура PARTITION вызывается не более n раз, выполняя при этом фиксированный объем работы и некоторое количество итераций цикла **for**. Каждая итерация цикла **for** выполняет строку 4. ■

Таким образом, наша цель заключается в том, чтобы вычислить величину X, т.е. полное количество сравнений, выполняемых при всех вызовах процедуры PARTITION. Не будем пытаться проанализировать, сколько сравнений производится при *каждом* вызове этой процедуры. Вместо этого получим общую оценку полного количества сравнений. Для этого необходимо понять, в каких случаях в алгоритме производится сравнение двух элементов массива, а в каких — нет. Для упрощения анализа переименуем элементы массива A как $z_1, z_2, \ldots, z_n$, где z_i — i-й наименьший по порядку элемент. Кроме того, определим множество $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$, которое содержит элементы, расположенные между элементами z_i и z_j включительно.

Когда в алгоритме выполняется сравнение элементов z_i и z_j? Прежде чем ответить на этот вопрос, заметим, что сравнение каждой пары элементов производится не более одного раза. Почему? Дело в том, что элементы сравниваются с опорным элементом, который никогда не используется в двух разных вызовах процедуры PARTITION. Таким образом, после некоторого вызова этой процедуры

используемый в качестве опорного элемент впоследствии не будет сравниваться с другими элементами.

Воспользуемся в нашем анализе индикаторными случайными величинами (см. раздел 5.2). Определим величину

$$X_{ij} = \mathrm{I}\{z_i \text{ сравнивается с } z_j\} ,$$

с помощью которой учитывается, произошло ли сравнение в течение работы алгоритма (но не в течение определенной итерации или определенного вызова процедуры PARTITION). Поскольку каждая пара элементов сравнивается не более одного раза, полное количество сравнений, выполняемых на протяжении работы алгоритма, легко выразить следующим образом:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Применяя к обеим частям этого выражения операцию вычисления математического ожидания и используя свойство ее линейности и лемму 5.1, получим

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ сравнивается с } z_j\} . \end{aligned} \qquad (7.2)$$

Осталось вычислить величину $\Pr\{z_i \text{ сравнивается с } z_j\}$. Предполагается, что опорные элементы выбираются случайным образом и независимо один от другого.

Полезно поразмышлять о том, когда два элемента *не* сравниваются. Рассмотрим в качестве входных данных для алгоритма быстрой сортировки множество, состоящее из целых чисел от 1 до 10 (в произвольном порядке), и предположим, что в качестве первого опорного элемента выбрано число 7. Тогда в результате первого вызова процедуры PARTITION все числа разбиваются на два множества: $\{1, 2, 3, 4, 5, 6\}$ и $\{8, 9, 10\}$. При этом элемент 7 сравнивается со всеми остальными. Очевидно, что ни одно из чисел, попавшее в первое подмножество (например, 2), больше не будет сравниваться ни с одним элементом второго подмножества (например, с 9).

В общем случае ситуация такая. Поскольку предполагается, что значения всех элементов различаются, при выборе в качестве опорного элемента такого x, что $z_i < x < z_j$, элементы z_i и z_j впоследствии сравниваться не будут. С другой стороны, если в качестве опорного элемент z_i выбран до любого другого элемента Z_{ij}, то он будет сравниваться с каждым элементом множества Z_{ij}, кроме

себя самого. Аналогичное утверждение можно сделать по поводу элемента z_j. В рассматриваемом примере значения 7 и 9 сравниваются, поскольку элемент 7 — первый из множества $Z_{7,9}$, выбранный в качестве опорного. По той же причине (поскольку элемент 7 — первый из множества $Z_{2,9}$, выбранный в качестве опорного) элементы 2 и 9 сравниваться не будут. Таким образом, элементы z_i и z_j сравниваются тогда и только тогда, когда первым в роли опорного в множестве Z_{ij} выбран один из них.

Теперь вычислим вероятность этого события. Перед тем как в множестве Z_{ij} будет выбран опорный элемент, все это множество является не разделенным, и любой его элемент с одинаковой вероятностью может стать опорным. Поскольку всего в этом множестве $j - i + 1$ элемент, а опорные элементы выбираются случайным образом и независимо один от другого, вероятность того, что какой-либо фиксированный элемент первым будет выбран в качестве опорного, равна $1/(j - i + 1)$. Таким образом, мы имеем

$$\begin{aligned}
\Pr\{z_i \text{ сравнивается с } z_j\} &= \Pr\{z_i \text{ или } z_j \text{ — первый опорный элемент,} \\
&\qquad \text{выбранный из } Z_{ij}\} \\
&= \Pr\{z_i \text{ — первый опорный элемент, выбранный} \\
&\qquad \text{из } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ — первый опорный элемент, выбранный} \\
&\qquad \text{из } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} . \qquad (7.3)
\end{aligned}$$

Вторая строка в приведенной выше цепочке равенств следует из того, что рассматриваемые события взаимоисключающие. Комбинируя уравнения (7.2) и (7.3), получаем

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

Эту сумму можно оценить, если воспользоваться заменой переменных ($k = j - i$) и границей для гармонического ряда (уравнение (А.7)):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}
\end{aligned}$$

$$= \sum_{i=1}^{n-1} O(\lg n)$$
$$= O(n \lg n) . \qquad (7.4)$$

Таким образом, можно сделать вывод, что при использовании процедуры RANDOMIZED-PARTITION математическое ожидание времени работы алгоритма быстрой сортировки различающихся по величине элементов равно $O(n \lg n)$.

Упражнения

7.4.1

Покажите, что в рекуррентном соотношении

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4.2

Покажите, что время работы быстрой сортировки в наилучшем случае равно $\Omega(n \lg n)$.

7.4.3

Покажите, что выражение $q^2 + (n - q - 1)^2$ достигает максимума на множестве значений $q = 0, 1, \ldots, n - 1$, когда $q = 0$ или $q = n - 1$.

7.4.4

Покажите, что ожидаемое время работы процедуры RANDOMIZED-QUICKSORT равно $\Omega(n \lg n)$.

7.4.5

На практике время работы алгоритма быстрой сортировки можно улучшить, воспользовавшись тем, что алгоритм сортировки по методу вставок быстро работает для "почти" отсортированных входных последовательностей. Для этого можно поступить следующим образом. Когда процедура быстрой сортировки начнет рекурсивно вызываться для обработки подмассивов, содержащих менее k элементов, в ней не будет производиться никаких действий, кроме выхода из процедуры. После возврата из процедуры быстрой сортировки, вызванной на самом высоком уровне, запускается алгоритм сортировки по методу вставок, на вход которого подается весь обрабатываемый массив. На этом процесс сортировки завершается. Докажите, что математическое ожидание времени работы такого алгоритма сортировки равно $O(nk + n \lg(n/k))$. Как следует выбирать значение k, исходя из теоретических и практических соображений?

7.4.6 ★

Рассмотрим следующую модификацию процедуры PARTITION. В ней случайным образом выбираются три элемента массива A, и разбиение массива выполняет-

ся по медиане выбранных элементов (т.е. по среднему из этих трех элементов). Найдите приближенную величину вероятности того, что в худшем случае разбиение будет произведено в отношении α к $(1-\alpha)$, как функцию от α в диапазоне $0 < \alpha < 1$.

Задачи

7.1. Корректность разбиения по Хоару

Представленная в этой главе версия процедуры PARTITION не является реализацией первоначально предложенного алгоритма. Ниже приведен исходный алгоритм разбиения, разработанный Ч.Э.Р. Хоаром (C.A.R. Hoare).

```
HOARE-PARTITION(A, p, r)
1  x = A[p]
2  i = p - 1
3  j = r + 1
4  while TRUE
5      repeat
6          j = j - 1
7      until A[j] ≤ x
8      repeat
9          i = i + 1
10     until A[i] ≥ x
11     if i < j
12         Обменять A[i] и A[j]
13     else return j
```

а. Продемонстрируйте, как работает процедура HOARE-PARTITION с массивом $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$. Для этого укажите, чему будут равны значения элементов массива и значения вспомогательных переменных после каждой итерации цикла **while** в строках 4–13.

В следующих трех заданиях предлагается привести аргументы в пользу корректности процедуры HOARE-PARTITION. В предположении, что подмассив $A[p..r]$ содержит как минимум два элемента, докажите следующие утверждения.

б. Индексы i и j принимают такие значения, что никогда не произойдет обращение к элементам массива A, находящимся за пределами подмассива $A[p..r]$.

в. По завершении процедуры HOARE-PARTITION она возвращает значение j, такое, что $p \leq j < r$.

г. По завершении процедуры HOARE-PARTITION каждый элемент подмассива $A[p..j]$ не превышает значений каждого элемента подмассива $A[j+1..r]$.

В описанной в разделе 7.1 процедуре PARTITION опорный элемент (которым изначально является элемент $A[r]$) отделяется от двух образованных с его помощью подмассивов. В процедуре же HOARE-PARTITION, напротив, опорный элемент (которым изначально является элемент $A[p]$) всегда помещается в один из двух полученных подмассивов: $A[p \mathinner{..} j]$ или $A[j+1 \mathinner{..} r]$. Поскольку $p \leq j < r$, это разбиение всегда нетривиально.

д. Перепишите процедуру QUICKSORT таким образом, чтобы в ней использовалась процедура HOARE-PARTITION.

7.2. Быстрая сортировка с равными элементами

Анализ ожидаемого времени работы рандомизированной быстрой сортировки в разделе 7.4.2 предполагает, что все значения элементов различны. В данной задаче мы рассмотрим, что произойдет в противном случае.

а. Предположим, что значения всех элементов одинаковы. Каким будет время работы рандомизированной быстрой сортировки в этом случае?

б. Процедура PARTITION возвращает индекс q, такой, что каждый элемент $A[p \mathinner{..} q-1]$ не превышает $A[q]$, а каждый элемент $A[q+1 \mathinner{..} r]$ больше $A[q]$. Модифицируйте процедуру PARTITION таким образом, чтобы получить процедуру PARTITION$'(A, p, r)$, которая переставляет элементы $A[p \mathinner{..} r]$ и возвращает два индекса q и t, где $p \leq q \leq t \leq r$, такие, что

- все элементы $A[q \mathinner{..} t]$ равны;
- каждый элемент $A[p \mathinner{..} q-1]$ меньше $A[q]$;
- каждый элемент $A[t+1 \mathinner{..} r]$ больше $A[q]$.

Как и процедура PARTITION, ваша процедура PARTITION$'$ должна иметь время работы $\Theta(r-p)$.

в. Модифицируйте процедуру RANDOMIZED-PARTITION так, чтобы она вызывала процедуру PARTITION$'$, и назовите новую процедуру именем RANDOMIZED-PARTITION$'$. Затем модифицируйте процедуру QUICKSORT таким образом, чтобы, в свою очередь, получить процедуру QUICKSORT$'(A, p, r)$, которая вызывает RANDOMIZED-PARTITION$'$ и рекурсивно работает только с теми частями, элементы которых не равны один другому.

г. Каким образом процедура QUICKSORT$'$ позволяет изменить анализ из раздела 7.4.2, чтобы избежать предположения о том, что все элементы различны?

7.3. Альтернативный анализ быстрой сортировки

Можно предложить альтернативный анализ рандомизированной быстрой сортировки, в ходе которого внимание сосредоточивается на математическом ожидании времени, которое требуется для выполнения каждого рекурсивного вызова процедуры QUICKSORT, а не на количестве производимых сравнений.

а. Докажите, что вероятность того, что какой-либо заданный элемент n-элементного массива будет выбран в качестве опорного, равна $1/n$. На основании этого факта определите индикаторную случайную величину

$$X_i = \mathrm{I}\left\{i\text{-й в порядке возрастания элемент выбран опорным}\right\} .$$

Что собой представляет величина $\mathrm{E}\left[X_i\right]$?

б. Пусть $T(n)$ — случайная величина, обозначающая время работы быстрой сортировки n-элементного массива. Докажите, что

$$\mathrm{E}\left[T(n)\right] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \qquad (7.5)$$

в. Покажите, что уравнение (7.5) можно представить в виде

$$\mathrm{E}\left[T(n)\right] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}\left[T(q)\right] + \Theta(n) . \qquad (7.6)$$

г. Покажите, что

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \qquad (7.7)$$

(*Указание:* разбейте сумму на две части, в одной из которых суммирование проводится по индексам $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$, а в другой — по индексам $k = \lceil n/2 \rceil, \ldots, n-1$.)

д. Покажите с помощью границ из (7.7), что решение рекуррентного соотношения (7.6) имеет вид $\mathrm{E}\left[T(n)\right] = \Theta(n \lg n)$. (*Указание:* с помощью подстановки покажите, что $\mathrm{E}\left[T(n)\right] \leq an \lg n$ для достаточно бóльших n и некоторой положительной константы a.)

7.4. Глубина стека быстрой сортировки

Алгоритм QUICKSORT из раздела 7.1 содержит два рекурсивных вызова самого себя. После того как процедура QUICKSORT вызывает процедуру PARTITION, она рекурсивно сортирует левый подмассив, а затем так же рекурсивно сортирует правый подмассив. Второй рекурсивный вызов в QUICKSORT в действительности не является необходимым; его можно избежать с помощью итеративной управляющей структуры. Этот метод, получивший название ***оконечной рекурсии*** (tail recursion), автоматически применяется хорошими компиляторами. Рассмотрите приведенную ниже версию быстрой сортировки, в которой имитируется оконечная рекурсия.

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
while p < r
    // Разбиение и сортировка левого подмассива
    q = PARTITION(A, p, r)
    TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
    p = q + 1
```

а. Докажите, что TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ корректно сортирует массив A.

Обычно компиляторы выполняют рекурсивные процедуры с помощью ***стека***, содержащего необходимую информацию, в том числе и значения параметров, применяющихся для каждого рекурсивного вызова. Информация о последнем вызове находится на вершине стека, а информация о начальном вызове — на его дне. При вызове процедуры информация ***заносится в стек*** (pushed), а при ее завершении — ***выталкивается*** (popped) из него. Поскольку предполагается, что массивы-параметры представлены указателями, при каждом обращении процедуры к стеку передается информация объемом $O(1)$. Глубина стека (stack depth) — это максимальная величина стекового пространства, которая используется в ходе вычисления.

б. Опишите сценарий, в котором глубина стека, необходимая для сортировки процедурой TAIL-RECURSIVE-QUICKSORT n-элементного массива, равна $\Theta(n)$.

в. Модифицируйте код процедуры TAIL-RECURSIVE-QUICKSORT так, чтобы необходимая для ее работы глубина стека в наихудшем случае была равна $\Theta(\lg n)$. Математическое ожидание времени работы алгоритма $O(n \lg n)$ должно при этом остаться неизменным.

7.5. Разбиение по медиане трех элементов

Один из способов улучшения процедуры RANDOMIZED-QUICKSORT заключается в том, чтобы производить разбиение по опорному элементу, определенному аккуратнее, чем путем случайного выбора. Один из распространенных подходов — метод ***медианы трех элементов***. Согласно этому методу в качестве опорного элемента выбирается медиана (средний элемент) подмножества, составленного из трех случайно выбранных в подмассиве элементов (см. упр. 7.4.6). В этой задаче предполагается, что все элементы входного массива $A[1..n]$ различны и что $n \geq 3$. Обозначим отсортированный выходной массив как $A'[1..n]$. Используя метод медианы трех элементов для выбора опорного элемента x, определим $p_i = \Pr\{x = A'[i]\}$.

а. Приведите точную формулу для величины p_i как функции от n и i для $i = 2, 3, \ldots, n-1$. (Заметим, что $p_1 = p_n = 0$.)

б. На сколько увеличится вероятность выбора в массиве $A[1..n]$ в качестве опорного элемента медианы $x = A'[\lfloor (n+1)/2 \rfloor]$, если используется не обычный

способ, а метод медианы? Чему равно предельное отношение этих вероятностей при $n \to \infty$?

в. Будем считать выбор опорного элемента $x = A'[i]$ "удачным", если $n/3 \leq i \leq 2n/3$. На сколько возрастает вероятность удачного выбора в методе медианы по сравнению с обычной реализацией? (*Указание:* используйте приближение суммы интегралом.)

г. Докажите, что при использовании метода медианы трех элементов время работы алгоритма быстрой сортировки, равное $\Omega(n \lg n)$, изменится только на постоянный множитель.

7.6. Нечеткая сортировка интервалов

Рассмотрим задачу сортировки, в которой отсутствуют точные сведения о значениях чисел. Вместо них для каждого числа на действительной числовой оси задается интервал, которому оно принадлежит. Таким образом, в нашем распоряжении имеется n закрытых интервалов, заданных в виде $[a_i, b_i]$, где $a_i \leq b_i$. Задача в том, чтобы произвести ***нечеткую сортировку*** (fuzzy-sort) этих интервалов, т.е. получить такую перестановку $\langle i_1, i_2, \ldots, i_n \rangle$ интервалов, чтобы для каждого $j = 1, 2, \ldots, n$ можно было найти значения $c_j \in [a_{i_j}, b_{i_j}]$, удовлетворяющие неравенствам $c_1 \leq c_2 \leq \cdots \leq c_n$.

а. Разработайте алгоритм нечеткой сортировки n интервалов. Общая структура предложенного алгоритма должна совпадать со структурой алгоритма, выполняющего быструю сортировку по левым границам интервалов (т.е. по значениям a_i). Однако время работы нового алгоритма должно быть улучшено за счет возможностей перекрывающихся интервалов. (По мере увеличения степени перекрытия интервалов задача их нечеткой сортировки все больше упрощается. В представленном алгоритме следует, насколько это возможно, воспользоваться указанным преимуществом.)

б. Докажите, что в общем случае математическое ожидание времени работы рассматриваемого алгоритма равно $\Theta(n \lg n)$, но если все интервалы перекрываются (т.е. если существует такое значение x, что $x \in [a_i, b_i]$) для всех i, то оно равно $\Theta(n)$. В предложенном алгоритме этот факт не следует проверять явно; его производительность должна улучшаться естественным образом по мере увеличения степени перекрытия.

Заключительные замечания

Процедуру быстрой сортировки впервые разработал Хоар (Hoare) [169]; предложенная им версия описана в задаче 7.1. Представленная в разделе 7.1 процедура PARTITION появилась благодаря Н. Ломуто (N. Lomuto). Содержащийся в разделе 7.4 анализ выполнен Авримом Блюмом (Avrim Blum). Хороший обзор литературы, посвященной особенностям реализации алгоритмов и их влиянию

на производительность, можно найти у Седжвика (Sedgewick) [303] и Бентли (Bentley) [42].

Мак-Илрой (McIlroy) [246] показал, как создать конструкцию, позволяющую получить массив, при обработке которого почти каждая реализация быстрой сортировки будет выполняться в течение времени $\Theta(n^2)$. Если реализация рандомизированная, то данная конструкция может создать данный массив на основании информации о том, каким образом в рандомизированном алгоритме быстрой сортировки осуществляется случайный выбор.

Глава 8. Сортировка за линейное время

Вы уже познакомились с несколькими алгоритмами, позволяющими выполнить сортировку n чисел за время $O(n \lg n)$. В алгоритмах сортировки слиянием и пирамидальной сортировки эта верхняя граница достигалась в наихудшем случае; в алгоритме быстрой сортировки она достигалась в среднем. Кроме того, для каждого из этих алгоритмов можно создать такую последовательность из n входных чисел, для которой алгоритм будет работать в течение времени $\Omega(n \lg n)$.

Все упомянутые алгоритмы обладают одним общим свойством: *при сортировке используется только сравнение входных элементов*. Назовем такие алгоритмы сортировки ***сортировкой сравнением*** (comparison sorts). Все описанные до настоящего момента алгоритмы сортировки принадлежат к данному типу.

В разделе 8.1 будет доказано, что при любой сортировке сравнением для обработки n элементов в наихудшем случае нужно произвести не менее $\Omega(n \lg n)$ сравнений. Таким образом, алгоритмы сортировки слиянием и пирамидальной сортировки асимптотически оптимальны, и не существует алгоритмов этого класса, которые бы работали быстрее и время выполнения которых отличалось бы больше, чем на постоянный множитель.

В разделах 8.2–8.4 рассматриваются три алгоритма сортировки: сортировка подсчетом (counting sort), поразрядная сортировка (radix sort) и карманная сортировка (bucket sort). Все эти алгоритмы выполняются в течение времени, линейно зависящего от количества элементов. Вряд ли стоит говорить о том, что в этих алгоритмах для определения правильного порядка элементов применяются операции, отличные от сравнений, а следовательно, нижняя граница $\Omega(n \lg n)$ к ним не применима.

8.1. Нижние границы для алгоритмов сортировки

В алгоритмах сортировки сравнением для получения информации о расположении элементов входной последовательности $\langle a_1, a_2, \ldots, a_n \rangle$ используются только попарные сравнения элементов. Другими словами, для определения взаимного порядка двух элементов a_i и a_j выполняется одна из проверок $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ или $a_i > a_j$. Значения самих элементов или иная информация о них не доступна никаким иным способом.

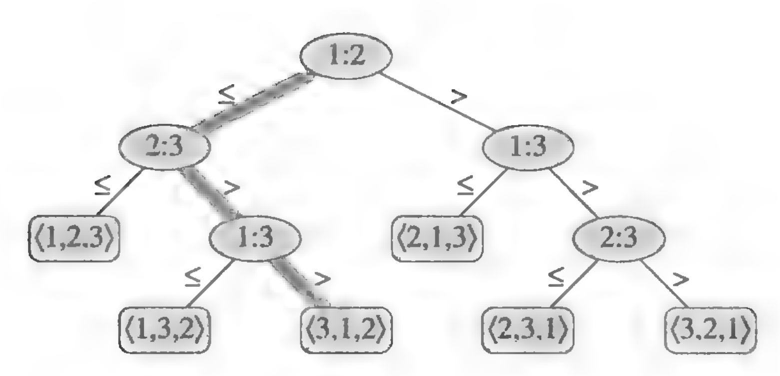


Рис. 8.1. Дерево решений для сортировки вставкой трех элементов. Внутренний узел, помеченный как $i{:}j$, указывает сравнение между a_i и a_j. Лист, помеченный перестановкой $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$, указывает упорядочение $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Выделенный путь указывает решения, принятые при сортировке входной последовательности $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; перестановка $\langle 3, 1, 2 \rangle$ в листе указывает, что отсортированным упорядочением является $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. Всего имеется $3! = 6$ возможных перестановок входных элементов, так что дерево решений должно иметь как минимум 6 листьев.

В этом разделе без потери общности предполагается, что все входные элементы различны. При этом операция сравнения $a_i = a_j$ становится бесполезной, так что можно считать, что никаких сравнений этого вида не производится. Заметим также, что операции $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ и $a_i < a_j$ эквивалентны в том смысле, что они дают одну и ту же информацию о взаимном расположении элементов a_i и a_j. Таким образом, можно считать, что все сравнения имеют вид $a_i \leq a_j$.

Модель дерева решений

Абстрактное рассмотрение алгоритмов сортировки сравнением можно производить с помощью деревьев решений (decision trees). Дерево решений — это полное бинарное дерево, в котором представлены операции сравнения элементов, производящиеся тем или иным алгоритмом сортировки, который обрабатывает входные данные заданного размера. Управляющие операции, перемещение данных и все другие аспекты алгоритма игнорируются. На рис. 8.1 показано дерево решений, которое соответствует представленному в разделе 2.1 алгоритму сортировки вставкой, обрабатывающему входную последовательность из трех элементов.

В дереве решений каждый внутренний узел помечен двухиндексной меткой $i : j$, где индексы i и j принадлежат интервалу $1 \leq i, j \leq n$, а n представляет собой количество элементов входной последовательности. Каждый лист также помечен — перестановкой $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (Детальный материал по перестановкам содержится в разделе В.1.) Выполнение алгоритма сортировки соответствует прохождению пути от корня дерева до одного из его листьев. Каждый внутренний узел соответствует выполнению сравнения $a_i \leq a_j$. Левое поддерево после этого сравнения определяет последующие сравнения после того, как мы выяснили, что $a_i \leq a_j$, а правое — в случае $a_i > a_j$. Дойдя до какого-нибудь из листьев, алгоритм сортировки устанавливает соответствующее этому листу упорядочение элементов $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Поскольку каждый корректный алгоритм сортировки должен быть способен произвести любую пе-

рестановку входных элементов, необходимое условие корректности сортировки сравнением заключается в том, что в листьях дерева решений должны размещаться все $n!$ перестановок n элементов и что из корня дерева к каждому его листу можно проложить путь, соответствующий одному из реальных вариантов работы сортировки сравнением. (Назовем такие листы достижимыми.) Таким образом, мы будем рассматривать только такие деревья решений, в которых каждая из перестановок представлена в виде достижимого листа.

Нижняя граница в наихудшем случае

Длина самого длинного пути от корня дерева решений к любому из его достижимых листьев соответствует количеству сравнений, которые выполняются в рассматриваемом алгоритме сортировки в наихудшем случае. Следовательно, количество сравнений, выполняемых в том или ином алгоритме сортировки сравнением в наихудшем случае, равно высоте его дерева решений. Поэтому нижняя граница высот для всех деревьев, в которых все перестановки представлены достижимыми листьями, является нижней оценкой времени работы для любого алгоритма сортировки сравнением. Эту оценку дает приведенная ниже теорема.

Теорема 8.1

Любой алгоритм сортировки сравнением в наихудшем случае требует выполнения $\Omega(n \lg n)$ сравнений.

Доказательство. Из приведенных выше рассуждений понятно, что для доказательства теоремы достаточно определить высоту дерева, в котором каждая перестановка представлена достижимым листом. Рассмотрим дерево решений высотой h с l достижимыми листьями, которое соответствует сортировке сравнением n элементов. Поскольку каждая из $n!$ перестановок входных элементов сопоставляется с одним из листьев, $n! \leq l$. Так как бинарное дерево высотой h имеет не более 2^h листьев, имеем

$$n! \leq l \leq 2^h ,$$

откуда после логарифмирования следует

$$\begin{aligned} h &\geq \lg(n!) && \text{(поскольку lg — строго возрастающая функция)} \\ &= \Omega(n \lg n) && \text{(согласно уравнению (3.19))} . \end{aligned}$$ ■

Следствие 8.2

Пирамидальная сортировка и сортировка слиянием являются асимптотически оптимальными сортировками сравнением.

Доказательство. Верхние границы $O(n \lg n)$ времени работы пирамидальной сортировки и сортировки слиянием совпадают с нижней границей $\Omega(n \lg n)$ для наихудшего случая из теоремы 8.1. ■

Упражнения

8.1.1

Чему равна наименьшая возможная глубина, на которой находится лист дерева решений сортировки сравнением?

8.1.2

Получите асимптотически точные границы для величины $\lg(n!)$, не используя приближение Стирлинга. Вместо этого воспользуйтесь для оценки суммы $\sum_{k=1}^{n} \lg k$ методами из раздела А.2.

8.1.3

Покажите, что не существует алгоритмов сортировки сравнением, время работы которых линейно по крайней мере для половины из $n!$ вариантов входных данных длиной n. Что можно сказать по поводу $1/n$-й части всех вариантов входных данных длиной n? По поводу $1/2^n$-й части?

8.1.4

Предположим, что у вас имеется последовательность, состоящая из n элементов. Входная последовательность состоит из n/k подпоследовательностей, в каждой из которых k элементов. Все элементы данной подпоследовательности меньше элементов следующей подпоследовательности и больше элементов предыдущей подпоследовательности. Таким образом, для сортировки всей n-элементной последовательности достаточно отсортировать k элементов в каждой из n/k подпоследовательностей. Покажите, что нижняя граница количества сравнений, необходимых для решения этой разновидности задачи сортировки, равна $\Omega(n \lg k)$. (*Указание:* просто скомбинировать нижние границы для отдельных подпоследовательностей недостаточно.)

8.2. Сортировка подсчетом

В ***сортировке подсчетом*** (counting sort) предполагается, что каждый из n входных элементов — целое число, принадлежащее интервалу от 0 до k, где k — некоторая целая константа. Если $k = O(n)$, то время работы алгоритма сортировки подсчетом равно $\Theta(n)$.

Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x определить количество элементов, которые меньше x. С помощью этой информации элемент x можно разместить в той позиции выходного массива, где он должен находиться. Например, если всего имеется 17 элементов, которые меньше x, то в выходной последовательности элемент x должен занимать 18-ю позицию. Если возможна ситуация, когда несколько элементов имеют одно и то же значение, эту схему нужно слегка модифицировать, поскольку все такие элементы не могут размещаться в одной и той же позиции.

В коде сортировки подсчетом предполагается, что входные данные представляют собой массив $A[1..n]$, и, таким образом, $A.length = n$. Требуются также два дополнительных массива: массив $B[1..n]$ хранит отсортированные выходные данные, а массив $C[0..k]$ предоставляет временное рабочее пространство.

```
COUNTING-SORT(A, B, k)
 1  Пусть C[0..k] — новый массив
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // Сейчас C[i] содержит количество элементов, равных i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i - 1]
 9  // Сейчас C[i] содержит количество элементов, не превышающих i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

Работа алгоритма сортировки подсчетом проиллюстрирована на рис. 8.2. После того как в цикле **for** в строках 2 и 3 массив C инициализирован нулевыми значениями, цикл **for** в строках 4 и 5 просматривает каждый входной элемент. Если значение входного элемента равно i, мы увеличиваем $C[i]$. Таким образом, после строки 5 элемент $C[i]$ содержит количество входных элементов, равных i, для каждого целого значения $i = 0, 1, \ldots, k$. В строках 7 и 8 для каждого $i = 0, 1, \ldots, k$ путем суммирования элементов массива C определяется, сколько входных элементов не превышают i.

Наконец в цикле **for** в строках 10–12 каждый элемент $A[j]$ помещается в надлежащую позицию выходного массива B. Если все n элементов различны, то при первом переходе к строке 10 для каждого элемента $A[j]$ в переменной $C[A[j]]$ хранится корректный индекс конечного положения этого элемента в выходном массиве, поскольку имеется $C[A[j]]$ элементов, меньших или равных $A[j]$. Поскольку разные элементы могут иметь одни и те же значения, помещая значение $A[j]$ в массив B, мы каждый раз уменьшаем $C[A[j]]$ на единицу. Благодаря этому следующий входной элемент, значение которого равно $A[j]$ (если таковой имеется), в выходном массиве размещается непосредственно перед элементом $A[j]$.

Сколько времени требуется для сортировки методом подсчета? Цикл **for** в строках 2 и 3 выполняется за время $\Theta(k)$, цикл **for** в строках 4 и 5 выполняется за время $\Theta(n)$, цикл **for** в строках 7 и 8 выполняется за время $\Theta(k)$ и цикл **for** в строках 10–12 выполняется за время $\Theta(n)$. Таким образом, общее время составляет $\Theta(k+n)$. На практике обычно сортировка подсчетом применяется, когда $k = O(n)$, и в этом случае общее время работы равно $\Theta(n)$.

Сортировка подсчетом превосходит нижнюю границу $\Omega(n \lg n)$, доказанную в разделе 8.1, поскольку не является сортировкой сравнением. Фактически нигде в коде не производится сравнение входных элементов. Вместо этого непосред-

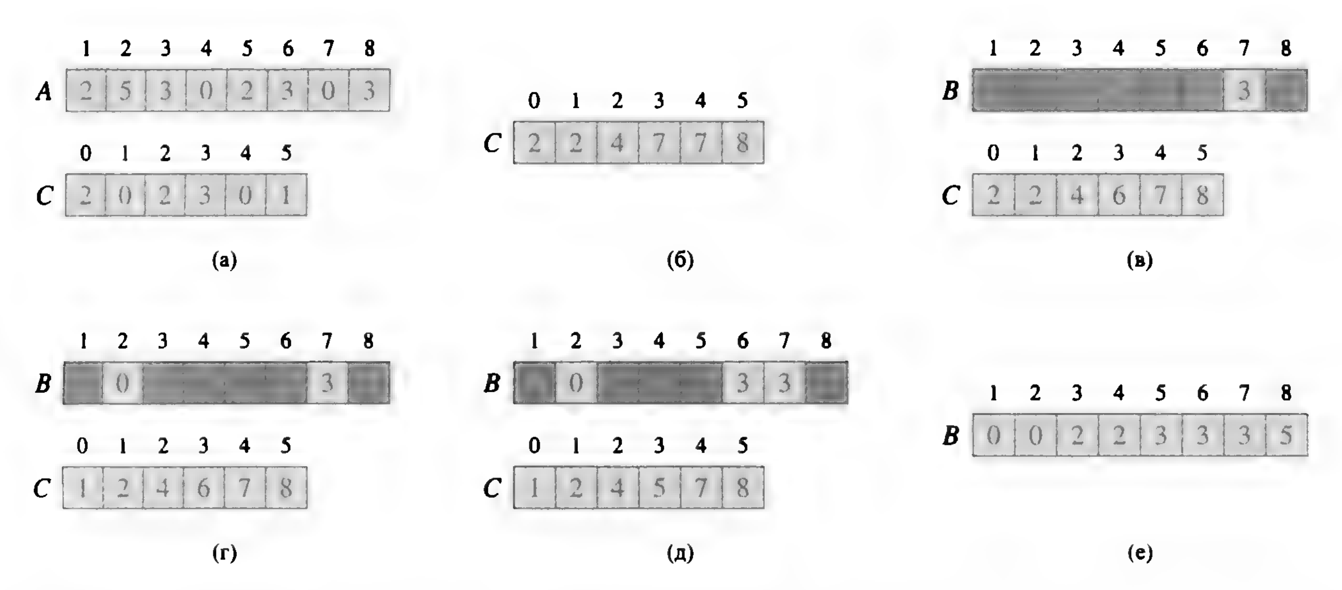


Рис. 8.2. Работа алгоритма COUNTING-SORT с входным массивом $A[1..8]$, где каждый элемент A представляет собой неотрицательное целое число, не превышающее $k = 5$. **(а)** Массив A и вспомогательный массив C после строки 5. **(б)** Массив C после строки 8. **(в)–(д)** Выходной массив B и вспомогательный массив C после одной, двух и трех итераций цикла в строках 10–12 соответственно. Заполненными являются только светло-серые элементы массива B. **(е)** Окончательный отсортированный выходной массив B.

ственно используются их значения, с помощью которых элементам сопоставляются конкретные индексы. Нижняя граница $\Omega(n \lg n)$ для сортировки сравнением неприменима при отказе от модели сортировки, использующей сравнения.

Важное свойство алгоритма сортировки подсчетом заключается в его ***устойчивости*** (stable): элементы с одним и тем же значением находятся в выходном массиве в том же порядке, что и во входном. Обычно свойство устойчивости важно только в ситуации, когда вместе сортируемые элементы имеют сопутствующие данные. Устойчивость, присущая сортировке подсчетом, важна еще и по другой причине: этот алгоритм часто используется в качестве подпрограммы при поразрядной сортировке. Как вы увидите в следующем разделе, устойчивость сортировки подсчетом критична для корректной работы поразрядной сортировки.

Упражнения

8.2.1

Используя рис. 8.2 в качестве образца, проиллюстрируйте обработку массива $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ процедурой COUNTING-SORT.

8.2.2

Докажите, что сортировка COUNTING-SORT устойчива.

8.2.3

Предположим, что заголовок цикла **for** (строка 10) в процедуре COUNTING-SORT переписан в следующем виде:

10 **for** $j = 1$ **to** $A.length$

8 Зак. 3726

Покажите, что алгоритм по-прежнему работает корректно. Устойчив ли модифицированный таким образом алгоритм?

8.2.4

Опишите алгоритм предварительной обработки n элементов, принадлежащих интервалу от 0 до k, после которого можно получить ответ на запрос о том, сколько из n входных элементов принадлежат отрезку $[a\,..\,b]$, за время $O(1)$. Алгоритм должен выполняться за время $\Theta(n + k)$.

8.3. Поразрядная сортировка

Поразрядная сортировка (radix sort) — это алгоритм, который использовался в машинах, предназначенных для сортировки перфокарт. Такие машины теперь можно найти разве что в музеях вычислительной техники. Перфокарты были разбиты на 80 столбцов, в каждом из которых в одной из 12 позиций можно было сделать отверстие. Сортировщик можно было механически "запрограммировать" таким образом, чтобы он проверял заданный столбец в каждой перфокарте, которая находится в колоде, и распределял перфокарты по 12 приемникам в зависимости от того, в какой позиции расположено отверстие. После этого оператор получал возможность извлечь перфокарты из всех приемников и разместить их так, чтобы сверху находились перфокарты с пробитым первым разрядом, за ними — перфокарты с пробитым вторым разрядом и т.д.

При записи десятичных цифр в каждом столбце используются лишь первые десять разрядов (остальные два разряда нужны для кодирования символов, отличных от цифр). Таким образом, число, состоящее из d цифр, занимает поле из d столбцов. Поскольку сортировщик за один раз может обработать только один столбец, для решения задачи о сортировке n перфокарт в порядке возрастания записанных на них d-значных чисел требуется разработать некий алгоритм сортировки.

Интуиция подсказывает способ сортировки, при котором выполняется сортировка по *старшей* цифре, а затем полученные стопки рекурсивно сортируются по следующим в порядке старшинства цифрам. К сожалению, в этом случае возникает большое количество промежуточных стопок перфокарт (после первой же сортировки по старшей цифре — десять стопок), за которыми нужно следить (см. упр. 8.3.5).

В алгоритме поразрядной сортировки поставленная задача решается способом, противоположным подсказываемому интуицией. Сначала производится сортировка по *младшей* цифре, после чего перфокарты снова объединяются в одну колоду, в которой сначала идут перфокарты из нулевого приемника, затем — из первого приемника, затем — из второго и т.д. После этого вся колода снова сортируется по предпоследней цифре, и перфокарты вновь собираются в одну стопку тем же образом. Процесс продолжается до тех пор, пока перфокарты не окажутся отсортированными по всем d цифрам. После этого перфокарты оказываются полностью

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

Рис. 8.3. Поразрядная сортировка списка из семи трехзначных чисел. В крайнем слева столбце показаны входные числа, а в последующих столбцах — последовательные состояния списка после его сортировки по цифрам, начиная с младшей. Серым цветом выделен текущий разряд, по которому производится сортировка, в результате чего получается следующий (расположенный справа) столбец.

отсортированными в порядке возрастания d-значных чисел. Таким образом, для сортировки перфокарт требуется лишь d проходов колоды. На рис. 8.3 показано, как поразрядная сортировка обрабатывает "колоду" из семи трехзначных чисел.

Важно, чтобы сортировка по цифрам того или иного разряда в этом алгоритме обладала устойчивостью. Сортировка, которая производится сортировщиком перфокарт, устойчива, но оператор должен также следить за тем, чтобы не перепутать порядок перфокарт после извлечения их из приемника. Это важно, несмотря на то что на всех перфокартах из одного и того же приемника в столбце, номер которого соответствует этапу обработки, стоит одна и та же цифра.

В типичных компьютерах, представляющих собой машины с произвольным доступом к памяти, поразрядная сортировка иногда применяется для приведения в порядок записей, ключи которых разбиты на несколько полей. Например, пусть нужно выполнить сортировку дат по трем ключам: год, месяц и день. Для этого можно было бы запустить алгоритм сортировки с функцией сравнения, в котором в двух заданных датах сначала сравнивались бы годы, при их совпадении сравнивались бы месяцы, а при совпадении и тех, и других сравнивались бы дни. Можно поступить и по-другому, т.е. выполнить трехкратную сортировку с помощью устойчивой процедуры: сначала — по дням, затем — по месяцам и наконец — по годам.

Код поразрядной сортировки прост и прямолинеен. В приведенной ниже процедуре предполагается, что каждый из n элементов массива A — это число, в котором всего d цифр, причем первая цифра стоит в самом младшем разряде, а цифра под номером d — в самом старшем разряде.

```
RADIX-SORT(A, d)
for i = 1 to d
    Выполнить устойчивую сортировку массива A по цифре i
```

Лемма 8.3

Пусть имеется n d-значных чисел, в которых каждая цифра принимает одно из k возможных значений. Тогда алгоритм RADIX-SORT позволяет выполнить корректную сортировку этих чисел за время $\Theta(d(n+k))$, если устойчивая сортировка, используемая данным алгоритмом, имеет время работы $\Theta(n+k)$.

Доказательство. Корректность поразрядной сортировки доказывается методом математической индукции по сортируемым столбцам (см. упр. 8.3.3). Анализ времени работы рассматриваемого алгоритма зависит от того, какой из методов устойчивой сортировки используется в качестве промежуточного алгоритма сортировки. Если каждая цифра принадлежит интервалу от 0 до $k - 1$ (и, таким образом, может принимать k возможных значений) и k не слишком велико, то оптимальным выбором является сортировка подсчетом. Для обработки каждой из d цифр n чисел понадобится время $\Theta(n + k)$, а все цифры будут обработаны алгоритмом поразрядной сортировки в течение времени $\Theta(d(n + k))$. ■

Если d — константа, а $k = O(n)$, то время работы алгоритма поразрядной сортировки линейно зависит от количества входных элементов. В общем случае мы получаем определенную степень свободы в выборе разбиения ключей на цифры.

Лемма 8.4

Пусть имеется n b-битовых чисел и произвольное натуральное число $r \leq b$. Алгоритм RADIX-SORT позволяет выполнить корректную сортировку этих чисел за время $\Theta((b/r)(n+2^r))$, если используемая им устойчивая сортировка имеет время работы $\Theta(n + k)$ для входных данных в диапазоне от 0 до k.

Доказательство. Для значения $r \leq b$ каждый ключ можно рассматривать как число, состоящее из $d = \lceil b/r \rceil$ цифр по r бит. Все цифры представляют собой целые числа в интервале от 0 до $2^r - 1$, поэтому можно воспользоваться алгоритмом сортировки подсчетом, в котором $k = 2^r - 1$ (например, 32-битовое слово можно рассматривать как число, состоящее из четырех 8-битовых цифр, так что $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, а $d = b/r = 4$). Каждый проход сортировки подсчетом занимает время $\Theta(n + k) = \Theta(n + 2^r)$, а всего выполняется d проходов, так что полное время работы алгоритма равно $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Выберем для двух заданных значений n и b такую величину $r \leq b$, которая бы минимизировала выражение $(b/r)(n + 2^r)$. Если $b < \lfloor \lg n \rfloor$, то для любого значения $r \leq b$ имеем $(n + 2^r) = \Theta(n)$. Таким образом, выбор $r = b$ приводит к асимптотически оптимальному времени работы $(b/b)(n + 2^b) = \Theta(n)$. Если же $b \geq \lfloor \lg n \rfloor$, то наилучшее время с точностью до постоянного множителя можно получить, выбрав $r = \lfloor \lg n \rfloor$. Это можно понять из следующих рассуждений. Если выбрать $r = \lfloor \lg n \rfloor$, то время работы алгоритма будет равно $\Theta(bn/\lg n)$. Если r увеличивается и превышает значение $\lfloor \lg n \rfloor$, то член 2^r в числителе возрастает быстрее, чем член r в знаменателе, так что увеличение r свыше $\lfloor \lg n \rfloor$ приводит ко времени работы алгоритма, равному $\Omega(bn/\lg n)$. Если же величина r уменьшается и становится меньше $\lfloor \lg n \rfloor$, то член b/r возрастает, а множитель $n + 2^r$ остается величиной порядка $\Theta(n)$.

Является ли поразрядная сортировка более предпочтительной, чем сортировка сравнением, например быстрая сортировка? Если $b = O(\lg n)$, как это часто бывает, и мы выбираем $r \approx \lg n$, то время работы алгоритма поразрядной сортировки равно $\Theta(n)$, что выглядит предпочтительнее ожидаемого времени выполнения быстрой сортировки $\Theta(n \lg n)$. Однако в этих Θ-выражениях совершенно

разные постоянные множители. Несмотря на то что для поразрядной сортировки n ключей может понадобиться меньше проходов, чем для их быстрой сортировки, каждый проход при поразрядной сортировке может длиться существенно дольше. Выбор подходящего алгоритма сортировки зависит от характеристик реализации алгоритма, от машины, на которой производится сортировка (например, при быстрой сортировке аппаратный кеш часто используется эффективнее, чем при поразрядной сортировке), а также от входных данных. Кроме того, в версии поразрядной сортировки, в которой в качестве промежуточного этапа используется устойчивая сортировка подсчетом, обработка элементов производится с привлечением дополнительной памяти, которая не нужна во многих алгоритмах сортировки сравнением со временем работы $\Theta(n \lg n)$. Таким образом, если в первую очередь нужно учитывать объем расходуемой памяти, может оказаться более предпочтительным использование алгоритма, в котором элементы обрабатываются на месте, например алгоритма быстрой сортировки.

Упражнения

8.3.1

Используя рис. 8.3 в качестве образца, проиллюстрируйте работу алгоритма RADIX-SORT со следующим списком английских слов: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3.2

Какие из перечисленных ниже алгоритмов сортировки устойчивы: сортировка вставкой, сортировка слиянием, пирамидальная сортировка, быстрая сортировка? Приведите простую схему, в результате применения которой любой алгоритм сортировки становится устойчивым. Оцените количество дополнительного времени и объем памяти, необходимые для реализации этой схемы.

8.3.3

Докажите методом математической индукции корректность работы поразрядной сортировки. Где в доказательстве используется предположение об устойчивости промежуточной сортировки?

8.3.4

Покажите, как выполнить сортировку n чисел, принадлежащих интервалу от 0 до $n^3 - 1$ за время $O(n)$.

8.3.5 ★

Определите точное количество проходов, которые понадобятся для сортировки d-значных десятичных чисел в наихудшем случае, если используется сортировка перфокарт, начинающаяся со старшей цифры. За каким количеством стопок перфокарт нужно будет следить оператору в наихудшем случае?

8.4. Карманная сортировка

В случае ***карманной сортировки*** (bucket sort) предполагается, что входные данные подчиняются равномерному закону распределения; время работы в среднем случае при этом оказывается равным $O(n)$. Карманная сортировка, как и сортировка подсчетом, работает быстрее, чем алгоритмы сортировки сравнением. Это происходит благодаря определенным предположениям о входных данных. Если при сортировке методом подсчета предполагается, что входные данные состоят из целых чисел, принадлежащих небольшому интервалу, то при карманной сортировке предполагается, что входные числа генерируются случайным процессом и равномерно распределены в интервале $[0, 1)$ (определение равномерного распределения можно найти в разделе В.2).

Карманная сортировка разбивает интервал $[0, 1)$ на n одинаковых интервалов, или ***карманов*** (buckets), а затем распределяет по этим карманам n входных чисел. Поскольку последние равномерно распределены в интервале $[0, 1)$, мы предполагаем, что в каждый из карманов попадет не очень много элементов. Чтобы получить выходную последовательность, нужно просто выполнить сортировку чисел в каждом кармане, а затем последовательно перечислить элементы каждого кармана.

При составлении кода карманной сортировки предполагается, что на вход подается массив A, состоящий из n элементов, и что величина каждого принадлежащего массиву элемента $A[i]$ удовлетворяет неравенству $0 \leq A[i] < 1$. Для работы нам понадобится вспомогательный массив связанных списков (карманов) $B[0\,..\,n-1]$; предполагается, что в нашем распоряжении имеется механизм поддержки таких списков. (Реализация основных операций при работе со связанным списком описана в разделе 10.2.)

```
BUCKET-SORT(A)
n = A.length
Пусть B[0..n-1] — новый массив
for i = 0 to n-1
    Сделать B[i] пустым списком
for i = 1 to n
    Вставить A[i] в список B[⌊nA[i]⌋]
for i = 0 to n-1
    Отсортировать список B[i] сортировкой вставкой
Соединить списки B[0], B[1], ..., B[n-1] в указанном порядке
```

На рис. 8.4 показана работа карманной сортировки с входным массивом из десяти чисел.

Чтобы понять, как работает этот алгоритм, рассмотрим два элемента — $A[i]$ и $A[j]$. Без потери общности можно предположить, что $A[i] \leq A[j]$. Поскольку $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, элемент $A[i]$ помещается либо в тот же карман, что и элемент $A[j]$, либо в карман с меньшим индексом. В первом случае элементы $A[i]$

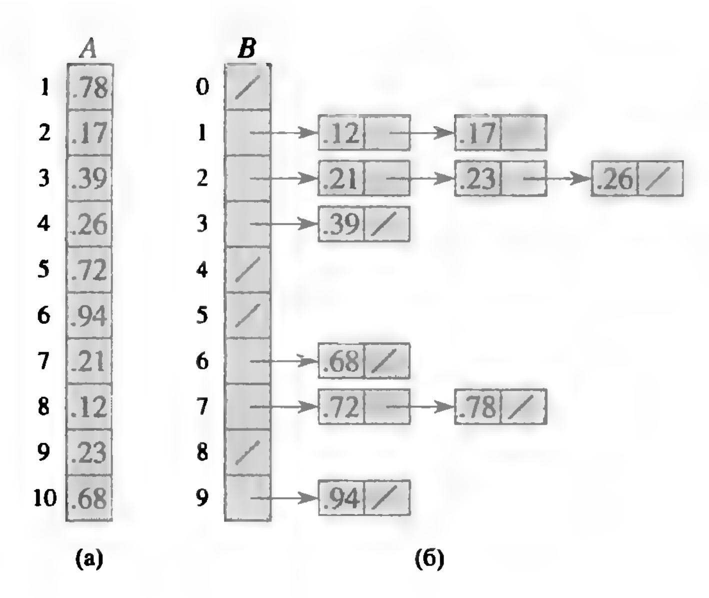


Рис. 8.4. Работа процедуры BUCKET-SORT для $n = 10$. **(а)** Входной массив $A[1..10]$. **(б)** Массив $B[0..9]$ отсортированных списков (карманов) после строки 8 алгоритма. В кармане i хранятся значения из полуоткрытого интервала $[i/10, (i+1)/10)$. Отсортированные выходные данные представляют собой конкатенацию списков $B[0], B[1], \ldots, B[9]$ в указанном порядке.

и $A[j]$ располагаются в нужном порядке благодаря циклу **for** в строках 7 и 8. Если же эти элементы попадут в разные карманы, то они разместятся в правильном порядке после выполнения строки 9. Таким образом, карманная сортировка работает корректно.

Чтобы проанализировать время работы алгоритма, заметим, что в наихудшем случае для выполнения всех его строк, кроме строки 8, требуется время $O(n)$. Остается просуммировать полное время, которое потребуется для n вызовов алгоритма сортировки вставкой (строка 8).

Чтобы оценить стоимость этих вызовов, введем случайную величину n_i, обозначающую количество элементов, попавших в карман $B[i]$. Поскольку время работы алгоритма сортировки вставкой является квадратичной функцией от количества входных элементов (см. раздел 2.2), время работы алгоритма карманной сортировки равно

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Теперь проанализируем время работы карманной сортировки в среднем случае, вычисляя математическое ожидание времени работы, где усреднение производится по входному распределению. Применяя операцию математического ожи-

дания к обеим частям и используя ее линейность, получаем

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(из линейности математического ожидания)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(согласно (B.22))} . \qquad (8.1)
\end{aligned}$$

Мы утверждаем, что

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \qquad (8.2)$$

для $i = 0, 1, \ldots, n-1$. Не удивительно, что каждому i-му карману соответствует одна и та же величина $\mathrm{E}[n_i^2]$, поскольку все элементы входного массива могут попасть в любой карман с равной вероятностью. Чтобы доказать уравнение (8.2), определим для каждого $i = 0, 1, \ldots, n-1$ и $j = 1, 2, \ldots, n$ индикаторную случайную величину

$$X_{ij} = \mathrm{I}\{A[j] \text{ попадает в карман } i\} .$$

Следовательно,

$$n_i = \sum_{j=1}^{n} X_{ij} .$$

Чтобы вычислить величину $\mathrm{E}[n_i^2]$, раскроем квадрат и перегруппируем слагаемые:

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right] , \qquad (8.3)
\end{aligned}$$

где последнее равенство следует из линейности математического ожидания. Отдельно вычислим обе суммы. Индикаторная случайная величина X_{ij} равна 1 с ве-

роятностью $1/n$ и 0 — в противном случае, поэтому получаем

$$\begin{aligned} \mathrm{E}\left[X_{ij}^2\right] &= 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} \; . \end{aligned}$$

При $k \neq j$ величины X_{ij} и X_{ik} независимы, а следовательно,

$$\begin{aligned} \mathrm{E}\left[X_{ij}X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \; . \end{aligned}$$

Подставив эти величины в уравнение (8.3), получим

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n} \; , \end{aligned}$$

что доказывает уравнение (8.2).

Используя это ожидаемое значение в уравнении (8.1), приходим к выводу, что время работы алгоритма карманной сортировки в среднем случае равно $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Такая зависимость может наблюдаться даже в том случае, когда входные элементы не подчиняются закону равномерного распределения. Если входные элементы обладают тем свойством, что сумма возведенных в квадрат размеров карманов линейно зависит от количества входных элементов, уравнение (8.1) утверждает, что карманная сортировка этих данных выполняется в течение времени, линейно зависящего от количества данных.

Упражнения

8.4.1

Используя рис. 8.4 в качестве образца, проиллюстрируйте работу алгоритма BUCKET-SORT над массивом $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4.2

Поясните, почему время работы карманной сортировки в наихудшем случае составляет $\Theta(n^2)$. Какое простое изменение следует внести в этот алгоритм, чтобы,

сохранив линейное время работы в среднем случае, добиться в наихудшем случае времени работы $O(n \lg n)$?

8.4.3

Пусть X — случайная величина, равная количеству выпадений орла при двух бросках правильной монеты. Чему равно $\mathrm{E}[X^2]$? $\mathrm{E}^2[X]$?

8.4.4 ★

Даны n точек в единичном круге, $p_i = (x_i, y_i)$, такие, что $0 < x_i^2 + y_i^2 \leq 1$ для $i = 1, 2, \ldots, n$. Предположим, что эти точки равномерно распределены, т.е. вероятность найти точку в любой области круга пропорциональна площади этой области. Разработайте алгоритм с временем работы $\Theta(n)$ в среднем случае для сортировки n точек по их расстояниям $d_i = \sqrt{x_i^2 + y_i^2}$ от центра. (*Указание:* спроектируйте размеры карманов в BUCKET-SORT, которые отражали бы равномерность распределения точек в единичном круге.)

8.4.5 ★

Функция распределения вероятности $P(x)$ случайной величины X определяется как $P(x) = \Pr\{X \leq x\}$. Предположим, что у нас есть список из n случайных величин $X_1, X_2, \ldots, X_n$ с непрерывной функцией распределения вероятности P, вычислимой за время $O(1)$. Разработайте алгоритм, сортирующий этот список за линейное время в среднем случае.

Задачи

8.1. Вероятностные нижние границы сортировки сравнением

В этой задаче мы докажем, что нижняя граница времени работы любого детерминистического или рандомизированного алгоритма сортировки сравнением при обработке n различных входных элементов равна $\Omega(n \lg n)$. Начнем с того, что рассмотрим детерминистическую сортировку сравнением A, которой соответствует дерево решений T_A. Предполагается, что все перестановки входных элементов A равновероятны.

а. Предположим, что каждый лист дерева T_A помечен вероятностью его достижения при заданном случайном наборе входных данных. Докажите, что ровно $n!$ листьям соответствует вероятность $1/n!$, а остальным — вероятность 0.

б. Пусть $D(T)$ — длина внешнего пути дерева решений T; другими словами, это сумма глубин всех листьев этого дерева. Пусть T — дерево решений с $k > 1$ листьями и пусть LT и RT — его левое и правое поддеревья. Покажите, что $D(T) = D(LT) + D(RT) + k$.

в. Пусть $d(k)$ — минимальная величина $D(T)$ среди всех деревьев решений T с $k > 1$ листьями. Покажите, что $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$. (*Указание:* рассмотрите дерево решений T с k листьями, на котором дости-

гается минимум. Обозначьте количество листьев в LT как i_0, а количество листьев в RT — как $k - i_0$.)

г. Докажите, что для данного значения $k > 1$ и i из диапазона $1 \leq i \leq k - 1$ функция $i \lg i + (k - i) \lg(k - i)$ достигает минимума при $i = k/2$. Выведите отсюда, что $d(k) = \Omega(k \lg k)$.

д. Докажите, что $D(T_A) = \Omega(n! \lg(n!))$, и выведите отсюда, что время сортировки n элементов в среднем случае составляет $\Omega(n \lg n)$.

Теперь рассмотрим *рандомизированную* сортировку B. Модель дерева решений можно обобщить таким образом, чтобы с ее помощью можно было рассматривать рандомизированные алгоритмы. Для этого в нее нужно включить узлы двух видов: обычные узлы сравнения и узлы "рандомизации", которые моделируют случайный выбор вида RANDOM$(1, r)$ в алгоритме B; такой узел имеет r дочерних узлов, каждый из которых в процессе выполнения алгоритма может быть выбран с равной вероятностью.

е. Покажите, что для любой рандомизированной сортировки сравнением B существует детерминистическая сортировка сравнением A, в которой ожидаемое количество сравнений не превышает количество сравнений в сортировке B.

8.2. Сортировка на месте за линейное время

Предположим, что имеется массив, содержащий n записей с сортируемыми данными, и что ключ каждой записи принимает значения 0 или 1. Алгоритм, предназначенный для сортировки такого набора записей, должен обладать некоторыми из трех перечисленных ниже характеристик.

1. Время работы алгоритма равно $O(n)$.
2. Алгоритм устойчив.
3. Сортировка выполняется на месте, т.е., кроме исходного массива, используется дополнительное пространство, не превышающее некоторой постоянной величины.

а. Разработайте алгоритм, удовлетворяющий критериям 1 и 2.

б. Разработайте алгоритм, удовлетворяющий критериям 1 и 3.

в. Разработайте алгоритм, удовлетворяющий критериям 2 и 3.

г. Может ли какой-либо из разработанных вами в пп. (а)–(в) алгоритмов использоваться в строке 2 процедуры RADIX-SORT, чтобы последняя могла сортировать n записей с b-битовыми ключами за время $O(bn)$? Поясните свой ответ.

д. Предположим, что n записей обладают ключами, значения которых находятся в интервале от 1 до k. Покажите, как можно модифицировать алгоритм сортировки подсчетом, чтобы обеспечить сортировку этих записей на месте за

время $O(n + k)$. В дополнение к входному массиву можно использовать дополнительную память объемом $O(k)$. Устойчив ли ваш алгоритм? (*Указание:* подумайте, как можно решить задачу для $k = 3$.)

8.3. Сортировка элементов переменной длины

а. Имеется массив целых чисел, причем различные элементы этого массива могут иметь разные количества цифр; однако общее количество цифр во *всех* числах равно n. Покажите, как выполнить сортировку этого массива за время $O(n)$.

б. Имеется массив строк, в котором различные строки могут иметь разную длину; однако общее количество символов во всех строках равно n. Покажите, как выполнить сортировку этого массива за время $O(n)$.

(Порядок сортировки в этой задаче определяется обычным алфавитным порядком, например $\mathtt{a} < \mathtt{ab} < \mathtt{b}$.)

8.4. Кувшины для воды

Предположим, что имеется n красных и n синих кувшинов для воды, которые различаются формой и объемом. Все красные кувшины могут вместить разное количество воды; то же самое относится и к синим кувшинам. Кроме того, каждому красному кувшину соответствует синий кувшин того же объема и наоборот.

Задача заключается в том, чтобы разделить кувшины на пары, в каждой из которых будут красный и синий кувшины одинакового объема. Для этого можно использовать такую операцию: сформировать пару кувшинов, в которых один будет синим, а второй — красным, наполнить красный кувшин водой, а затем перелить ее в синий кувшин. Эта операция позволит узнать, какой из кувшинов больше или является ли их объем одинаковым. Предположим, что для выполнения такой операции требуется одна единица времени. Необходимо сформулировать алгоритм, в котором для разделения кувшинов на пары производилось бы минимальное количество сравнений. Напомним, что непосредственно сравнивать два красных или два синих кувшина нельзя.

а. Опишите детерминистический алгоритм, в котором разделение кувшинов на пары производилось бы с помощью $\Theta(n^2)$ сравнений.

б. Докажите, что нижняя граница количества сравнений, которые должен выполнить алгоритм, предназначенный для решения этой задачи, равна $\Omega(n \lg n)$.

в. Разработайте рандомизированный алгоритм, математическое ожидание количества сравнений в котором было бы равно $O(n \lg n)$, и докажите корректность этой границы. Чему равно количество сравнений в этом алгоритме в наихудшем случае?

8.5. Сортировка в среднем

Предположим, что вместо сортировки массива нам нужно, чтобы его элементы возрастали в среднем. Точнее говоря, n-элементный массив A называется ***k*-отсортированным**, если для всех $i = 1, 2, \ldots, n-k$ выполняется соотношение

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

а. Что представляет собой 1-отсортированный массив?

б. Приведите пример перестановки чисел $1, 2, \ldots, 10$, являющейся 2-отсортированной, но *не* отсортированной.

в. Докажите, что n-элементный массив k-отсортирован тогда и только тогда, когда для всех $i = 1, 2, \ldots, n-k$ справедливо соотношение $A[i] \leq A[i+k]$.

г. Разработайте алгоритм, который выполняет k-сортировку n-элементного массива за время $O(n \lg(n/k))$.

Можно также найти нижнюю границу для времени, необходимого для получения k-отсортированного массива, если k — константа.

д. Покажите, что k-сортировку массива длиной n можно выполнить за время $O(n \lg k)$. (*Указание:* воспользуйтесь решением упр. 6.5.9.)

е. Покажите, что если k — константа, то для k-сортировки n-элементного массива потребуется время $\Omega(n \lg n)$. (*Указание:* воспользуйтесь решением предыдущего задания вместе с нижней границей для алгоритмов сортировки сравнением.)

8.6. Нижняя граница объединения отсортированных списков

Часто возникает задача объединения двух отсортированных списков. Эта процедура используется в качестве подпрограммы MERGE в разделе 2.3.1. В настоящей задаче мы покажем, что для количества сравнений, необходимых для объединения двух n-элементных отсортированных списков в наихудшем случае, существует нижняя граница, равная $2n-1$ сравнениям.

Сначала с помощью дерева решений покажем, что нижняя граница количества сравнений равна $2n - o(n)$.

а. Вычислите для заданных $2n$ чисел количество возможных способов их разделения на два отсортированных списка, каждый с n элементами.

б. Покажите с помощью дерева решений и вашего ответа к п. (а), что в любом алгоритме, корректно объединяющем два отсортированных списка, выполняется не менее $2n - o(n)$ сравнений.

Теперь мы докажем наличие несколько более точной границы $2n-1$.

в. Покажите, что если два элемента, которые в объединенном списке будут расположены последовательно один за другим, принадлежат разным подлежащим объединению спискам, то в ходе объединения эти элементы обязательно придется сравнить.

г. С помощью решения предыдущего пункта задачи покажите, что нижняя граница для количества сравнений, которые производятся при объединении двух отсортированных списков, равна $2n - 1$.

8.7. 0-1-лемма сортировки и сортировка столбцами

Операция ***сравнения с обменом*** над двумя элементами массива $A[i]$ и $A[j]$, где $i < j$, имеет следующий вид.

```
COMPARE-EXCHANGE(A, i, j)
if A[i] > A[j]
    Обменять A[i] с A[j]
```

Мы знаем, что после операции сравнения с обменом $A[i] \leq A[j]$.

Алгоритм сортировки ***сравнением с обменом без запоминания*** работает с помощью исключительно последовательности операций сравнения с обменом. Индексы сравниваемых позиций в последовательности должны быть определены заранее, и хотя они могут зависеть от количества сортируемых элементов, они не могут зависеть ни от сортируемых значений, ни от результатов предшествующих операций сравнения. Например, вот как сортировка вставкой выражается как алгоритм сортировки сравнением с обменом без запоминания.

```
INSERTION-SORT(A)
for j = 2 to A.length
    for i = j − 1 downto 1
        COMPARE-EXCHANGE(A, i, i + 1)
```

0-1-лемма сортировки предоставляет мощное средство для доказательства того, что алгоритм сортировки сравнением с обменом без запоминания дает корректные результаты. Она гласит, что если этот алгоритм корректно сортирует все входные последовательности, состоящие из нулей и единиц, то он корректно сортирует все входные данные с произвольными значениями.

Докажем 0-1-лемму сортировки путем доказательства обратной к ней: если алгоритм сортировки сравнением с обменом без запоминания некорректно работает для входных данных с произвольными значениями, то он будет некорректно сортировать и некоторые 0-1-входные данные. Предположим, что алгоритм сортировки сравнением с обменом без запоминания X неверно сортирует массив $A[1..n]$. Пусть $A[p]$ — наименьшее значение в A, которое алгоритм X помещает в неверную позицию, и пусть $A[q]$ представляет собой значение, которое алгоритм X перемещает в позицию, в которую должно было попасть значение $A[p]$. Опреде-

лим массив $B[1\,..\,n]$ из 0 и 1 следующим образом.

$$B[i] = \begin{cases} 0\,, & \text{если } A[i] \le A[p]\,, \\ 1\,, & \text{если } A[i] > A[p]\,. \end{cases}$$

а. Докажите, что $A[q] > A[p]$, так что $B[p] = 0$ и $B[q] = 1$.

б. Для завершения доказательства 0-1-леммы сортировки докажите, что алгоритм X неверно сортирует массив B.

Теперь используем 0-1-лемму сортировки для доказательства того, что конкретный алгоритм сортировки работает корректно. Алгоритм ***сортировки столбцами*** (columnsort) работает с прямоугольным массивом из n элементов. Массив имеет r строк и s столбцов (так что $n = rs$), удовлетворяющих следующим ограничениям:

- r должно быть четным;
- s должно быть делителем r;
- $r \ge 2s^2$.

По завершении сортировки столбцами массив оказывается отсортированным ***в порядке старшинства столбцов***: при чтении столбцов сверху вниз и слева направо элементы монотонно увеличиваются.

Сортировка столбцами работает за восемь шагов независимо от значения n. Все нечетные шаги одинаковы: отдельная сортировка каждого столбца. Каждый четный шаг представляет собой фиксированную перестановку. Вот эти шаги.

1. Отсортировать каждый столбец.
2. Транспонировать массив, но вновь привести его к виду с r строками и s столбцами. Другими словами, преобразовать крайний слева столбец поочередно в r/s верхних строк; затем преобразовать следующий столбец в следующие r/s строк и т.д.
3. Отсортировать каждый столбец.
4. Выполнить инверсию перестановки, выполняемой на шаге 2.
5. Отсортировать каждый столбец.
6. Сдвинуть верхнюю половину каждого столбца в нижнюю, а нижнюю — в верхнюю половину столбца справа. Оставить верхнюю половину крайнего слева столбца пустой, а нижнюю часть крайнего справа столбца сдвинуть в верхнюю половину нового крайнего справа столбца, оставляя пустой нижнюю половину этого нового столбца.
7. Отсортировать каждый столбец.
8. Выполнить обращение перестановки, выполняемой на шаге 6.

10	14	5
8	7	17
12	1	6
16	9	11
4	15	2
18	3	13

(а)

4	1	2
8	3	5
10	7	6
12	9	11
16	14	13
18	15	17

(б)

4	8	10
12	16	18
1	3	7
9	14	15
2	5	6
11	13	17

(в)

1	3	6
2	5	7
4	8	10
9	13	15
11	14	17
12	16	18

(г)

1	4	11
3	8	14
6	10	17
2	9	12
5	13	16
7	15	18

(д)

1	4	11
2	8	12
3	9	14
5	10	16
6	13	17
7	15	18

(е)

	5	10	16
	6	13	17
	7	15	18
1	4	11	
2	8	12	
3	9	14	

(ж)

	4	10	16
	5	11	17
	6	12	18
1	7	13	
2	8	14	
3	9	15	

(з)

1	7	13
2	8	14
3	9	15
4	10	16
5	11	17
6	12	18

(и)

Рис. 8.5. Шаги сортировки столбцами. **(а)** Входной массив с шестью строками и тремя столбцами. **(б)** После сортировки столбцов на шаге 1. **(в)** После транспонирования и изменения формы на шаге 2. **(г)** После сортировки столбцов на шаге 3. **(д)** После выполнения шага 4, обращающего перестановку на шаге 2. **(е)** После сортировки столбцов на шаге 5. **(ж)** После сдвига на полстолбца на шаге 6. **(з)** После сортировки столбцов на шаге 7. **(и)** После выполнения шага 8, обращающего перестановку на шаге 6. Теперь массив отсортирован в порядке старшинства столбцов.

На рис. 8.5 показан пример выполнения шагов сортировки столбцами при $r = 6$ и $s = 3$. (Этот пример работает, несмотря даже на то, что в нем нарушено требование $r \geq 2s^2$.)

в. Докажите, что можно рассматривать сортировку столбцами как алгоритм сортировки сравнением с обменом без запоминания, даже если мы не знаем, какой метод сортировки используется на нечетных шагах.

Хотя в корректность алгоритма сортировки сравнением с обменом без запоминания может быть трудно поверить, для доказательства этого факта можно воспользоваться 0-1-леммой сортировки. Эта лемма применима, поскольку сортировку столбцами можно рассматривать как алгоритм сортировки сравнением с обменом без запоминания. Пара определений поможет вам применить 0-1-лемму сортировки. Мы говорим, что область массива ***чистая*** (clean), если она содержит только нули или только единицы. В противном случае область может содержать как нули, так и единицы, и является ***грязной*** (dirty). Впредь будем считать, что входной массив содержит только нули и единицы и что его можно рассматривать как массив с r строками и s столбцами.

г. Докажите, что после шагов 1–3 массив состоит из некоторых чистых строк из нулей вверху, нескольких чистых строк с единицами внизу и не более чем из s грязных строк между ними.

д. Докажите, что после шага 4 массив при чтении в порядке старшинства столбцов начинается с чистой области, состоящей из нулей, а заканчивается чистой областью, состоящей из единиц, и имеет посредине грязную область не более чем из s^2 элементов.

е. Докажите, что шаги 5–8 генерируют полностью отсортированный 0-1-выход. Сделайте отсюда вывод о том, что сортировка столбцами корректно сортирует все входные данные с произвольными значениями.

ж. Теперь предположим, что s не является делителем r. Докажите, что после шагов 1–3 массив состоит из нескольких чистых строк из нулей вверху, нескольких чистых строк из единиц внизу и не более чем из $2s - 1$ грязных строк между ними. Насколько бóльшим по сравнению с r должно быть значение s, чтобы сортировка столбцами работала корректно, когда s не является делителем r?

з. Предложите простое изменение шага 1, которое позволяет обеспечить требование $r \geq 2s^2$ даже при s, не являющемся делителем r, и докажите, что это изменение не влияет на корректность работы алгоритма.

Заключительные замечания

Использовать модель дерева решений при изучении алгоритмов сортировки сравнением впервые предложили Форд (Ford) и Джонсон (Johnson) [109]. В томе *Искусства программирования* Кнута (Knuth), посвященном сортировке [210][1], описываются различные разновидности задачи сортировки, включая приведенную здесь теоретико-информационную нижнюю границу для сложности сортировки. Полное исследование нижних границ сортировки с помощью обобщений модели дерева решений было проведено Бен-Ором (Ben-Or) [38].

Согласно Кнуту сортировка подсчетом впервые была предложена Х.Г. Сьювардом (H.H. Seward) в 1954 году; ему также принадлежит идея объединения сортировки подсчетом и поразрядной сортировки. Оказывается, что поразрядная сортировка, начинающаяся с самой младшей значащей цифры, — по сути "народный" алгоритм, широко применявшийся операторами механических машин, предназначенных для сортировки перфокарт. Как утверждает Кнут, первая ссылка на этот метод появилась в документе, составленном Л.Д. Комри (L.J. Comrie) и опубликованном в 1929 году, где описывается счетно-перфорационное оборудование. Карманная сортировка используется с 1956 года, с того времени, когда И.Д. Исаак (E.J. Isaac) и Р.К. Синглтон (R.C. Singleton) предложили основную идею этого метода [187].

[1]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

Мунро (Munro) и Раман (Raman) [261] предложили устойчивый алгоритм сортировки, который в наихудшем случае выполняет $O(n^{1+\epsilon})$ сравнений, где $0 < \epsilon \leq 1$ — произвольная константа. Несмотря на то что в алгоритмах, время выполнения которых равно $O(n \lg n)$, выполняется меньше сравнений, в алгоритме Мунро и Рамана данные перемещаются $O(n)$ раз, и он выполняет сортировку на месте.

Случай сортировки n b-битовых чисел за время $o(n \lg n)$ рассматривался многими исследователями. Было получено несколько положительных результатов, в каждом из которых незначительно менялись предположения о вычислительной модели, а также накладываемые на алгоритм ограничения. Во всех случаях предполагалось, что память компьютера разделена на адресуемые b-битовые слова. Фредман (Fredman) и Виллард (Willard) [114] первыми предложили использовать дерево слияний (fusion tree) и выполнять с его помощью сортировку n целых чисел за время $O(n \lg n / \lg \lg n)$. Впоследствии эта граница была улучшена Андерссоном (Andersson) [16] до $O(n\sqrt{\lg n})$. В этих алгоритмах применяются операция умножения, а также некоторые предварительно вычисляемые константы. Андерссон, Хейгерап (Hagerup), Нильсон (Nilsson) и Раман [17] показали, как выполнить сортировку n чисел за время $O(n \lg \lg n)$, не прибегая при этом к умножению, однако этот метод требует дополнительной памяти, объем которой может неограниченно увеличиваться с ростом n. С помощью мультипликативного хеширования объем этой памяти можно уменьшить до величины $O(n)$, но при этом граница для наихудшего случая $O(n \lg \lg n)$ становится границей для математического ожидания времени работы. Обобщив экспоненциальные деревья поиска Андерссона [16], Торуп (Thorup) [333] сформулировал алгоритм сортировки, выполняющийся в течение времени $O(n(\lg \lg n)^2)$. В этом алгоритме не используется ни умножение, ни рандомизация, а объем необходимой дополнительной памяти линейно зависит от количества элементов. Дополнив эти методы новыми идеями, Хан (Han) [157] улучшил границу времени работы до $O(n \lg \lg n \lg \lg \lg n)$. Несмотря на то что упомянутые алгоритмы стали важным теоретическим достижением, все они чрезвычайно сложные, и в данный момент представляется маловероятным, чтобы они могли практически составить конкуренцию существующим алгоритмам сортировки.

Сортировка столбцами из задачи 8.7 разработана Лейтоном (Leighton) [226].

Глава 9. Медианы и порядковые статистики

Будем называть i-й ***порядковой статистикой*** (order statistic) множества, состоящего из n элементов, i-й элемент в порядке возрастания. Например, ***минимум*** такого множества — это первая порядковая статистика ($i = 1$), а его ***максимум*** — это n-я порядковая статистика ($i = n$). ***Медиана*** (median) неформально обозначает середину множества. Если n нечетное, то медиана единственная, и ее индекс равен $i = (n+1)/2$; если же n четное, то медианы две, и их индексы равны $i = n/2$ и $i = n/2+1$. Таким образом, независимо от четности n, медианы располагаются при $i = \lfloor (n+1)/2 \rfloor$ (***нижняя медиана*** (lower median)) и $i = \lceil (n+1)/2 \rceil$ (***верхняя медиана*** (upper median)). Однако для простоты в этой книге термин "медиана" относится к нижней медиане.

Данная глава посвящена задаче выбора i-й порядковой статистики в множестве, состоящем из n различных чисел. Для удобства предположим, что все числа множества различны, хотя почти все, что мы будем делать, обобщается на случай, когда некоторые значения в множестве повторяются. Формально ***задачу выбора*** (selection problem) можно определить следующим образом.

Вход. Множество A, состоящее из n (различных) чисел, и число $1 \leq i \leq n$.

Выход. Элемент $x \in A$, превышающий по величине ровно $i-1$ других элементов множества A.

Задачу выбора можно решить за время $O(n \lg n)$. Для этого достаточно выполнить сортировку элементов с помощью пирамидальной сортировки или сортировки слиянием, а затем просто извлечь элемент выходного массива с индексом i. Однако в этой главе представлены более быстрые алгоритмы.

В разделе 9.1 рассматривается задача о выборе минимального и максимального элементов множества. Больший интерес представляет общая задача выбора, которая исследуется в двух последующих разделах. В разделе 9.2 анализируется применяющийся на практике рандомизированный алгоритм, ожидаемое время работы которого составляет $O(n)$ в предположении, что все элементы различны. В разделе 9.3 приведен алгоритм, представляющий более теоретический интерес, время работы которого достигает величины $O(n)$ в наихудшем случае.

9.1. Минимум и максимум

Сколько сравнений необходимо для того, чтобы найти минимальный элемент в n-элементном множестве? Для этой величины легко найти верхнюю границу, равную $n - 1$ сравнениям: мы по очереди проверяем каждый элемент множества и следим за тем, какой из них является минимальным на данный момент. В представленной ниже процедуре предполагается, что исследуется множество, состоящее из элементов массива A, где $A.length = n$.

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Очевидно, что для поиска максимального элемента также понадобится не более $n - 1$ сравнений.

Является ли представленный выше алгоритм оптимальным? Да, поскольку можно доказать, что нижняя граница для задачи определения минимума также равна $n - 1$ сравнений. Любой алгоритм, предназначенный для определения минимального элемента множества, можно представить в виде турнира, в котором принимают участие все элементы. Каждое сравнение — это поединок между двумя элементами, в котором побеждает элемент с меньшей величиной. Важное наблюдение заключается в том, что каждый элемент, кроме победителя, должен потерпеть поражение хотя бы в одном поединке. Таким образом, для определения минимума понадобится $n - 1$ сравнений, и алгоритм MINIMUM является оптимальным по отношению к количеству производимых в нем сравнений.

Одновременный поиск минимума и максимума

В некоторых приложениях возникает необходимость найти как минимальный, так и максимальный элементы множества. Например, графической программе может понадобиться выполнить масштабирование множества координат (x, y) таким образом, чтобы они совпали по размеру с прямоугольной областью экрана или другого устройства вывода. Для этого сначала нужно определить максимальное и минимальное значения координат.

Совершенно очевидно, как найти минимум и максимум в n-элементном множестве, производя при этом $\Theta(n)$ сравнений; при этом алгоритм будет асимптотически оптимальным. Достаточно просто выполнить независимый поиск минимального и максимального элементов. Для выполнения каждой подзадачи понадобится $n - 1$ сравнений, что в сумме составит $2n - 2$ сравнений.

Однако на самом деле для одновременного определения минимума и максимума достаточно не более $3\lfloor n/2 \rfloor$ сравнений. Для этого необходимо следить за тем, какой из проверенных на данный момент элементов минимальный, а какой —

максимальный. Вместо того чтобы отдельно сравнивать каждый входной элемент с текущим минимумом и максимумом (для чего пришлось бы на каждый элемент израсходовать по два сравнения), мы будем обрабатывать пары элементов. Образовав пару входных элементов, сначала сравним их *один с другим*, а затем меньший элемент пары будем сравнивать с текущим минимумом, а больший — с текущим максимумом. Таким образом, для каждой пары элементов понадобится по три сравнения.

Способ начального выбора текущего минимума и максимума зависит от четности количества элементов в множестве n. Если n нечетно, мы выбираем из множества один из элементов и считаем его значение одновременно и минимумом, и максимумом; остальные элементы обрабатываем парами. Если же n четно, то выбираем два первых элемента и путем сравнения определяем, значение какого из них будет минимумом, а какого — максимумом. Остальные элементы обрабатываем парами, как и в предыдущем случае.

Проанализируем, чему равно полное число сравнений. Если n нечетно, то нужно будет выполнить $3\lfloor n/2 \rfloor$ сравнений. Если же n четно, то выполняется одно начальное сравнение, а затем — еще $3(n-2)/2$ сравнений, что в сумме дает общее количество сравнений, равное $3n/2 - 2$. Таким образом, в обоих случаях полное количество сравнений не превышает $3\lfloor n/2 \rfloor$.

Упражнения

9.1.1

Покажите, что для поиска второго в порядке возрастания элемента в наихудшем случае достаточно $n + \lceil \lg n \rceil - 2$ сравнений. (*Указание:* найдите заодно и наименьший элемент.)

9.1.2 ★

Докажите, что в наихудшем случае для поиска максимального и минимального среди n чисел необходимо выполнить $\lceil 3n/2 \rceil - 2$ сравнений. (*Указание:* рассмотрите вопрос о том, сколько чисел являются потенциальными кандидатами на роль максимума или минимума, и определите, как на это количество влияет каждое сравнение.)

9.2. Выбор в течение линейного ожидаемого времени

Общая задача выбора оказывается более сложной, чем простая задача поиска минимума. Однако, как это ни удивительно, время решения обеих задач в асимптотическом пределе ведет себя одинаково — как $\Theta(n)$. В данном разделе вниманию читателя представляется алгоритм типа "разделяй и властвуй" RANDOMIZED-SELECT, предназначенный для решения задачи выбора. Этот алгоритм разработан по аналогии с алгоритмом быстрой сортировки, который рассматривался в главе 7. Как и в алгоритме быстрой сортировки, в алгоритме

RANDOMIZED-SELECT используется идея рекурсивного разбиения входного массива. Однако в отличие от алгоритма быстрой сортировки, в котором рекурсивно обрабатываются обе части разбиения, алгоритм RANDOMIZED-SELECT работает лишь с одной частью. Это различие проявляется в результатах анализа обоих алгоритмов: если математическое ожидание времени работы алгоритма быстрой сортировки равно $\Theta(n \lg n)$, то ожидаемое время работы алгоритма RANDOMIZED-SELECT равно $\Theta(n)$, в предположении, что все элементы входного множества различны.

В алгоритме RANDOMIZED-SELECT используется процедура RANDOMIZED-PARTITION, впервые представленная в разделе 7.3. Таким образом, подобно процедуре RANDOMIZED-QUICKSORT, RANDOMIZED-SELECT — это рандомизированный алгоритм, поскольку его поведение частично определяется выводом генератора случайных чисел. Приведенный ниже код процедуры RANDOMIZED-SELECT возвращает i-й в порядке возрастания элемент массива $A[p \mathinner{..} r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k           // Ответом является опорное значение
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Процедура RANDOMIZED-PARTITION работает следующим образом. В строке 1 выполняется проверка базового случая рекурсии, когда подмассив $A[p \mathinner{..} r]$ состоит из единственного элемента. В этом случае i должно быть равно 1, и мы просто возвращаем в строке 2 $A[p]$ как i-й в порядке возрастания элемент. В противном случае в строке 3 вызывается процедура RANDOMIZED-PARTITION, которая разбивает массив $A[p \mathinner{..} r]$ на два (возможно, пустых) подмассива $A[p \mathinner{..} q-1]$ и $A[q+1 \mathinner{..} r]$, таких, что величина каждого элемента $A[p \mathinner{..} q-1]$ не превышает значения $A[q]$, которое, в свою очередь, меньше любого из значений элементов $A[q+1 \mathinner{..} r]$. Как и в алгоритме быстрой сортировки, элемент $A[q]$ называется ***опорным*** (pivot). В строке 4 процедуры RANDOMIZED-SELECT вычисляется количество элементов k подмассива $A[p \mathinner{..} q]$, т.е. количество элементов, попадающих в нижнюю часть разбиения плюс один опорный элемент. Затем в строке 5 проверяется, является ли элемент $A[q]$ i-м в порядке возрастания элементом. Если это так, то он возвращается в строке 6. В противном случае в алгоритме определяется, в каком из двух подмассивов содержится i-й в порядке возрастания элемент: в подмассиве $A[p \mathinner{..} q-1]$ или $A[q+1 \mathinner{..} r]$. Если $i < k$, то нужный элемент находится в нижней части разбиения, и он рекурсивно выбирается из соответствующего подмассива в строке 8. Если же $i > k$, то нужный элемент находится в верхней части разбиения. Поскольку нам уже известны k значений, которые меньше

i-го в порядке возрастания элемента массива $A[p\,..\,r]$ (это элементы подмассива $A[p\,..\,q]$), искомый элемент является $(i-k)$-м в порядке возрастания элементом подмассива $A[q+1\,..\,r]$, который рекурсивно ищется в строке 9. Создается впечатление, что в представленном коде допускаются рекурсивные обращения к подмассивам, содержащим 0 элементов, но в упр. 9.2.1 предлагается показать, что это невозможно.

Время работы алгоритма RANDOMIZED-SELECT в наихудшем случае равно $\Theta(n^2)$, причем даже для поиска минимума. Дело в том, что фортуна может от нас отвернуться, и разбиение всегда будет производиться относительно наибольшего из оставшихся элементов, а само разбиение выполняется за время $\Theta(n)$. Однако алгоритм имеет линейное ожидаемое время работы, а поскольку он рандомизированный, никакие специально выбранные входные данные не могут гарантированно привести к наихудшему поведению алгоритма.

Чтобы проанализировать ожидаемое время работы алгоритма RANDOMIZED-SELECT, будем рассматривать время работы над массивом $A[p\,..\,r]$ из n элементов как случайную величину, которую обозначим как $T(n)$, так что мы можем получить верхнюю границу $\mathrm{E}[T(n)]$ следующим образом. Процедура RANDOMIZED-PARTITION равновероятно возвращает любой элемент в качестве опорного. Следовательно, для каждого k, такого, что $1 \le k \le n$, подмассив $A[p\,..\,q]$ содержит k элементов (все они не превышают опорный) с вероятностью $1/n$. Определим для $k = 1, 2, \ldots, n$ индикаторные случайные величины X_k, где

$$X_k = \mathrm{I}\{\text{подмассив } A[p\,..\,q] \text{ содержит ровно } k \text{ элементов}\} .$$

В предположении, что все элементы различны, имеем

$$\mathrm{E}[X_k] = 1/n . \tag{9.1}$$

В момент вызова процедуры RANDOMIZED-SELECT и выбора в качестве опорного элемента $A[q]$ заранее неизвестно, будет ли получен правильный ответ, после чего работа алгоритма сразу же прекратится, или произойдет рекурсивное обращение к подмассиву $A[p\,..\,q-1]$ либо подмассиву $A[q+1\,..\,r]$. Это будет зависеть от того, где будет расположен искомый элемент относительно элемента $A[q]$. В предположении монотонного неубывания функции $T(n)$ необходимое для рекурсивного вызова процедуры RANDOMIZED-SELECT время можно ограничить сверху временем, необходимым для вызова этой процедуры с входным массивом максимально возможного размера. Другими словами, для получения верхней границы предполагается, что искомый элемент всегда попадает в ту часть разбиения, где больше элементов. При конкретном вызове процедуры RANDOMIZED-SELECT индикаторная случайная величина X_k принимает значение 1 только для одного значения k, а при всех других k эта величина равна 0. Если $X_k = 1$, то размеры двух подмассивов, к которым может произойти рекурсивное обращение, равны

$k-1$ и $n-k$. Таким образом, получаем следующее рекуррентное соотношение:

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \;. \end{aligned}$$

Вычисляя математическое ожидание, получаем:

$$\begin{aligned} &\mathrm{E}\,[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}\,[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(в силу линейности математического ожидания)} \\ &= \sum_{k=1}^{n} \mathrm{E}\,[X_k] \cdot \mathrm{E}\,[T(\max(k-1, n-k))] + O(n) && \text{(согласно (B.24))} \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}\,[T(\max(k-1, n-k))] + O(n) && \text{(согласно (9.1))} \;. \end{aligned}$$

Применяя уравнение (B.24), мы основывались на независимости случайных величин X_k и $T(\max(k-1, n-k))$. В упр. 9.2.2 предлагается доказать это предположение.

Рассмотрим выражение $\max(k-1, n-k)$. Мы имеем

$$\max(k-1, n-k) = \begin{cases} k-1\;, & \text{если } k > \lceil n/2 \rceil\;, \\ n-k\;, & \text{если } k \le \lceil n/2 \rceil\;. \end{cases}$$

Если n четно, то каждое слагаемое от $T(\lceil n/2 \rceil)$ до $T(n-1)$ появляется в сумме ровно дважды. Если же n нечетно, то дважды появляются все слагаемые, кроме $T(\lfloor n/2 \rfloor)$, которое добавляется лишь один раз. Таким образом, имеем следующее соотношение:

$$\mathrm{E}\,[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}\,[T(k)] + O(n) \;.$$

Методом подстановок покажем, что $\mathrm{E}\,[T(n)] = O(n)$. Предположим, что $\mathrm{E}\,[T(n)] \le cn$ для некоторой константы c, удовлетворяющей начальным условиям рекуррентного соотношения. Кроме того, предположим, что для n, меньших какой-то константы, справедливо $T(n) = O(1)$; эта константа будет найдена позже. Выберем также константу a, такую, чтобы функция, соответствующая слагаемому $O(n)$ (и описывающая нерекурсивную составляющую времени работы данного алгоритма), была ограничена сверху величиной an для всех $n > 0$.

С помощью этой гипотезы индукции получаем

$$
\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2\rfloor-1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\lfloor n/2\rfloor}{2}\right) + an \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4-3n/2+2}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) .
\end{aligned}
$$

Чтобы завершить доказательство, нужно показать, что для достаточно больших n последнее выражение не превышает величину cn или (что равносильно) что выполняется неравенство $cn/4 - c/2 - an \ge 0$. Добавив к обеим частям этого неравенства $c/2$ и умножив их на n, получим неравенство $n(c/4 - a) \ge c/2$. Если константа c выбрана таким образом, что $c/4 - a > 0$, т.е. $c > 4a$, то обе части приведенного выше соотношения можно разделить на $c/4 - a$, что дает нам следующий результат:

$$
n \ge \frac{c/2}{c/4-a} = \frac{2c}{c-4a} .
$$

Таким образом, если предположить, что для всех $n < 2c/(c-4a)$ справедливо $T(n) = O(1)$, то $\mathrm{E}[T(n)] = O(n)$. Итак, мы приходим к выводу, что для поиска любой порядковой статистики (в частности, медианы) в предположении, что все элементы различны по величине, требуется ожидаемое время, линейно зависящее от количества входных элементов.

Упражнения

9.2.1

Покажите, что процедуре RANDOMIZED-SELECT никогда не передается в качестве параметра массив с нулевым количеством элементов.

9.2.2

Докажите, что индикаторная случайная величина X_k и величина $T(\max(k - 1, n - k))$ независимы.

9.2.3

Напишите итеративную версию процедуры RANDOMIZED-SELECT.

9.2.4

Предположим, что для выбора минимального элемента массива $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$ используется процедура RANDOMIZED-SELECT. Опишите последовательность разделений, соответствующих наихудшей производительности этой процедуры.

9.3. Алгоритм выбора с линейным временем работы в наихудшем случае

Рассмотрим теперь алгоритм выбора, время работы которого в наихудшем случае равно $O(n)$. Подобно алгоритму RANDOMIZED-SELECT, алгоритм SELECT находит нужный элемент путем рекурсивного разбиения входного массива. Однако в основе этого алгоритма заложена идея, которая заключается в том, чтобы *гарантировать* хорошее разбиение массива. В алгоритме SELECT используется детерминистическая процедура PARTITION, которая применяется при быстрой сортировке (см. раздел 7.1). Эта процедура модифицирована таким образом, чтобы одним из ее входных параметров был элемент, относительно которого производится разбиение.

Для определения во входном массиве, содержащем $n > 1$ элементов, i-го в порядке возрастания элемента в алгоритме SELECT выполняются описанные далее шаги. Если $n = 1$, то процедура SELECT просто возвращает единственное входное значение, как i-е в порядке возрастания.

1. Все n элементов входного массива разбиваются на $\lfloor n/5 \rfloor$ групп по 5 элементов и одну группу, содержащую оставшиеся $n \bmod 5$ элементов (впрочем, эта группа может оказаться пустой).

2. Сначала методом сортировки вставкой сортируется каждая из $\lceil n/5 \rceil$ групп (содержащих не более 5 элементов), а затем в каждом отсортированном списке, состоящем из элементов групп, выбирается медиана.

3. Путем рекурсивного использования процедуры SELECT определяется медиана x множества из $\lceil n/5 \rceil$ медиан, найденных на шаге 2. (Если этих медиан окажется четное количество, то, согласно принятому соглашению, переменной x будет присвоено значение нижней медианы.)

4. С помощью модифицированной версии процедуры PARTITION входной массив делится относительно медианы медиан x. Пусть число k на единицу превышает количество элементов, попавших в нижнюю часть разбиения. Тогда x — k-й

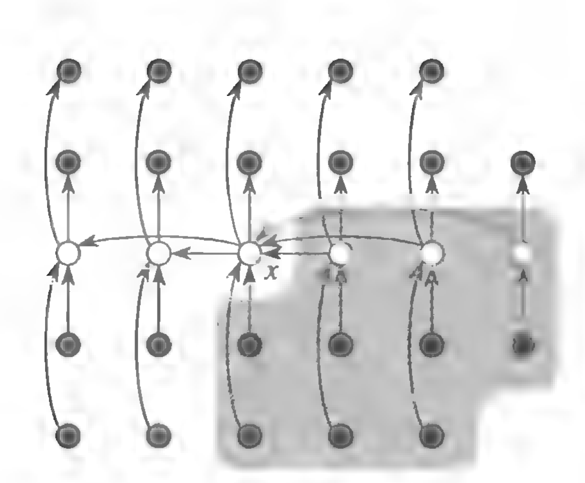


Рис. 9.1. Анализ алгоритма SELECT. На рисунке n элементов представлены в виде маленьких кружков, а каждая группа из 5 элементов — отдельным столбцом. Медианы групп обозначены белыми кружками, а медиана медиан x снабжена соответствующей меткой. (При определении медианы четного количества элементов используется нижняя медиана.) Стрелки проведены в направлении от бóльших элементов к меньшим. Из рисунка видно, что в каждой полной группе из 5 элементов, расположенной справа от x, содержится по 3 элемента, превышающих по величине x, а в каждой полной группе из 5 элементов, расположенной слева от x, содержится по 3 элемента, меньших по величине, чем x. Элементы, которые превышают x, выделены серым фоном.

в порядке возрастания элемент, и в верхнюю часть разбиения попадает $n - k$ элементов.

5. Если $i = k$, то возвращается значение x. В противном случае процедура SELECT вызывается рекурсивно, и с ее помощью выполняется поиск i-го в порядке возрастания элемента в нижней части, если $i < k$, или $(i - k)$-го в порядке возрастания элемента в верхней части, если $i > k$.

Чтобы проанализировать время работы процедуры SELECT, сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент x. Разобраться в этой бухгалтерии поможет рис. 9.1. Как минимум половина медиан, найденных на шаге 2, больше или равны медиане медиан x[1]. Таким образом, как минимум $\lceil n/5 \rceil$ групп содержат по 3 элемента, превышающих величину x, за исключением одной группы, в которой меньше пяти элементов (такая группа существует, если n не делится на 5 нацело), и еще одной группы, содержащей сам элемент x. Не учитывая эти две группы, приходим к выводу, что количество элементов, величина которых превышает x, равно как минимум

$$3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

Аналогично имеется не менее $3n/10 - 6$ элементов, величина которых меньше x. Таким образом, процедура SELECT рекурсивно вызывается на шаге 5 не более чем для $7n/10 + 6$ элементов.

---

[1] В силу предположения о том, что все числа различаются, каждая из этих медиан больше или меньше x, за исключением самой медианы x.

Теперь можно сформулировать рекуррентное соотношение для времени работы алгоритма SELECT (обозначим его как $T(n)$) в наихудшем случае. Для выполнения шагов 1, 2 и 4 требуется время $O(n)$ (шаг 2 состоит из $O(n)$ вызовов сортировки вставкой для множеств размером $O(1)$). Выполнение шага 3 занимает время $T(\lceil n/5 \rceil)$, а выполнение шага 5 — время не более $T(7n/10+6)$ (предполагается, что функция T монотонно неубывающая). Сделаем, на первый взгляд, необоснованное предположение о том, что для обработки любого входного массива, количество элементов которого меньше 140, требуется время $O(1)$ (вскоре нам раскроется магия константы 140). Таким образом мы получаем рекуррентное соотношение

$$T(n) \le \begin{cases} O(1) , & \text{если } n < 140 , \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) , & \text{если } n \ge 140 . \end{cases}$$

Покажем методом подстановки, что время работы, описываемое этим соотношением, линейно зависит от количества входных элементов. Точнее говоря, покажем, что для некоторой достаточно большой константы c и для всех $n > 0$ выполняется неравенство $T(n) \le cn$. Начнем с предположения, что это неравенство выполняется для некоторой достаточно большой константы c и для всех $n < 140$; это предположение действительно выполняется, если константа c выбрана достаточно большой. Кроме того, выберем константу a таким образом, чтобы функция, соответствующая представленному выше слагаемому $O(n)$ (которое описывает нерекурсивную составляющую времени работы алгоритма) для всех $n > 0$ была ограничена сверху величиной an. Подставив эту гипотезу индукции в правую часть рекуррентного соотношения, получим

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10+6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) . \end{aligned}$$

Это выражение не превышает величину cn, если выполняется неравенство

$$-cn/10 + 7c + an \le 0 . \tag{9.2}$$

При $n > 70$ неравенство (9.2) эквивалентно неравенству $c \ge 10a(n/(n-70))$. Поскольку мы предположили, что $n \ge 140$, то $n/(n-70) \le 2$, а значит, если выбрать $c \ge 20a$, то будет удовлетворяться неравенство (9.2). (Заметим, что в константе 140 нет ничего особенного; вместо нее можно было бы взять любое другое целое число, превышающее 70, после чего соответствующим образом нужно было бы выбрать константу c.) Таким образом, в наихудшем случае время работы алгоритма SELECT линейно зависит от количества входных элементов.

Как и в алгоритмах сортировки сравнением, которые рассматривались в разделе 8.1, в алгоритмах SELECT и RANDOMIZED-SELECT информация о взаимном расположении элементов извлекается только путем их сравнения. Как было дока-

зано в главе 8, в модели сравнений сортировка выполняется в течение времени $\Omega(n \lg n)$, даже в среднем (см. задачу 8.1). В этой же главе был рассмотрен алгоритм сортировки, время работы которого линейно зависит от количества сортируемых элементов, но в нем делаются дополнительные предположения относительно входных данных. Для работы представленных в настоящей главе алгоритмов выбора, время работы которых такое же, напротив, никаких дополнительных предположений не требуется. К этим алгоритмам не применима нижняя граница $\Omega(n \lg n)$, поскольку задача выбора в них решается без сортировки. Таким образом, решение задачи выбора путем сортировки и индексирования, представленное во введении к данной главе, асимптотически неэффективное.

Упражнения

9.3.1

В алгоритме SELECT все входные элементы делятся на группы по 5 элементов. Было бы время работы этого алгоритма линейным, если бы входной массив делился на группы по 7 элементов? Докажите, что время работы алгоритма не будет линейным, если в каждой группе будет по 3 элемента.

9.3.2

Проанализируйте алгоритм SELECT и покажите, что при $n \geq 140$ как минимум $\lceil n/4 \rceil$ элементов превышают по величине медиану медиан x и как минимум $\lceil n/4 \rceil$ элементов меньше x.

9.3.3

Покажите, каким образом можно выполнить быструю сортировку за время $O(n \lg n)$ в наихудшем случае, считая, что все элементы различны.

9.3.4 ★

Предположим, что в алгоритме, предназначенном для поиска среди n элементов i-го в порядке возрастания элемента, применяется только операция сравнения. Покажите, что с помощью этого алгоритма можно также найти $i - 1$ наименьших элементов и $n - i$ наибольших элементов, не выполняя никаких дополнительных сравнений.

9.3.5

Предположим, что у нас имеется подпрограмма поиска медиан, которая представляет собой “черный ящик” и время работы которой линейно зависит от количества входных элементов. Приведите пример алгоритма с линейным временем работы, с помощью которого задача выбора решалась бы для произвольной порядковой статистики.

9.3.6

По определению k-ми ***квантилями*** (quantiles) n-элементного множества называются $k - 1$ порядковых статистик, разбивающих это отсортированное множество на k одинаковых по размеру подмножеств (с точностью до одного элемента).

Сформулируйте алгоритм, который бы выводил список k-х квантилей множества за время $O(n \lg k)$.

9.3.7

Опишите алгоритм, который для заданного множества S, состоящего из n различных чисел, и положительной целой константы $k \leq n$ определял бы k ближайших соседей медианы множества S, принадлежащих этому множеству. Время работы алгоритма должно быть равно $O(n)$.

9.3.8

Пусть $X[1..n]$ и $Y[1..n]$ — два массива, каждый из которых содержит по n элементов, расположенных в отсортированном порядке. Разработайте алгоритм, в котором поиск медианы всех $2n$ элементов, содержащихся в массивах X и Y, выполнялся бы за время $O(\lg n)$.

9.3.9

Профессор работает консультантом в нефтяной компании, которая планирует провести магистральный трубопровод от восточного до западного края нефтяного месторождения с n скважинами. От каждой скважины к магистральному трубопроводу кратчайшим путем, в направлении на север или на юг, проведены рукава (рис. 9.2). Каким образом профессор может выбрать оптимальное расположение трубопровода (т.е. такое, при котором общая длина всех рукавов была бы минимальной) по заданным координатам скважин (x, y)? Покажите, что это можно сделать в течение времени, линейно зависящего от количества скважин.

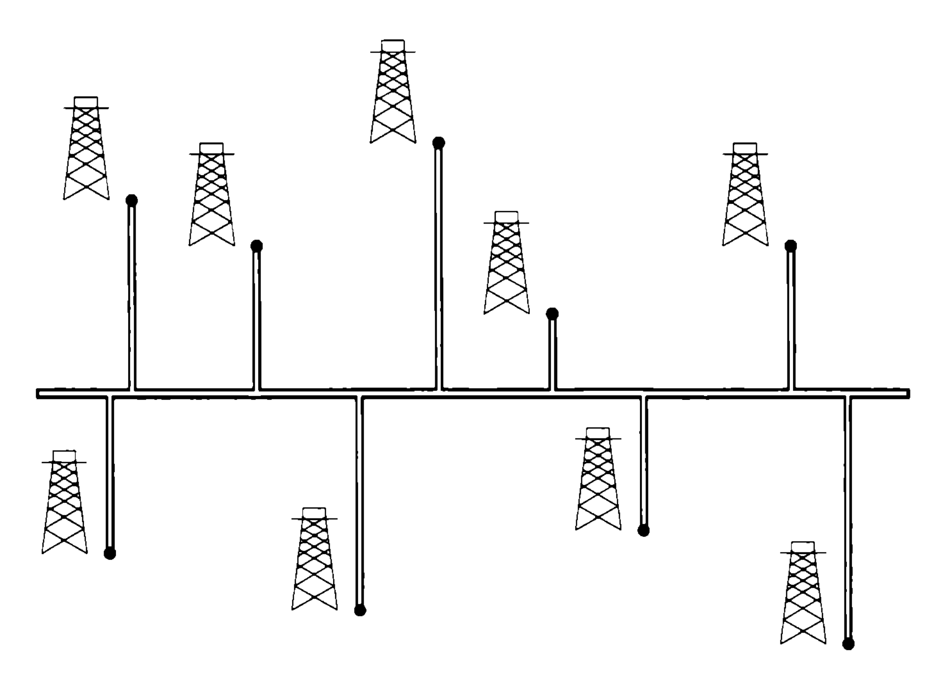


Рис. 9.2. Определение положения нефтепровода, при котором общая длина поперечных рукавов будет минимальной

Задачи

9.1. Наибольшие i чисел в порядке сортировки

Пусть имеется множество из n чисел, в котором с помощью алгоритма, основанного на сравнениях, нужно найти i наибольших элементов, расположенных в порядке сортировки. Сформулируйте алгоритм, в котором реализовывался бы каждый из перечисленных ниже методов с наилучшим возможным асимптотическим временем работы в наихудшем случае. Проанализируйте зависимость времени работы этих алгоритмов от n и i.

а. Отсортируйте эти числа и выведите i наибольших.

б. Создайте из чисел невозрастающую очередь с приоритетами и i раз вызовите процедуру EXTRACT-MAX.

в. Найдите с помощью алгоритма порядковой статистики i-й по порядку наибольший элемент, произведите разбиение относительно него и выполните сортировку i наибольших чисел.

9.2. Взвешенная медиана

Пусть имеется n различных элементов $x_1, x_2, \ldots, x_n$, для которых заданы положительные веса $w_1, w_2, \ldots, w_n$, такие, что $\sum_{i=1}^{n} w_i = 1$. ***Взвешенной (нижней) медианой*** (weighted (lower) median) называется элемент x_k, удовлетворяющий неравенствам

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

и

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2} \, .$$

Например, если элементами являются $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ и каждый элемент имеет вес, равный значению самогó элемента (т.е. $w_i = x_i$ для $i = 1, 2, \ldots, 7$), то медианой является 0.1, а взвешенной медианой — 0.2.

а. Докажите, что обычная медиана элементов $x_1, x_2, \ldots, x_n$ равна их взвешенной медиане, если все веса равны $w_i = 1/n$, $i = 1, 2, \ldots, n$.

б. Покажите, как вычислить взвешенную медиану n элементов с помощью сортировки, чтобы время вычисления в наихудшем случае было равно $O(n \lg n)$.

в. Покажите, как с помощью алгоритма, в котором поиск медианы производится за линейное время (подобного описанному в разделе 9.3 алгоритму SELECT), вычислить взвешенную медиану в течение времени, которое в наихудшем случае равно $\Theta(n)$.

Задача о размещении почтового отделения формулируется следующим образом. Имеется n точек $p_1, p_2, \ldots, p_n$, которым соответствуют веса $w_1, w_2, \ldots, w_n$. Нужно найти точку p (это не обязательно должна быть одна из входных точек), в которой минимизируется сумма $\sum_{i=1}^{n} w_i\, d(p, p_i)$, где $d(a, b)$ — расстояние между точками a и b.

г. Докажите, что взвешенная медиана — это наилучшее решение одномерной задачи об оптимальном размещении почтового отделения. В этой задаче точки представляют собой действительные числа, а расстояние между точками a и b определяется как $d(a, b) = |a - b|$.

д. Найдите наилучшее решение двумерной задачи об оптимальном размещении почтового отделения. В этой задаче точки задаются парами координат (x, y), а расстояние между точками $a = (x_1, y_1)$ и $b = (x_2, y_2)$ представляет собой ***манхэттенское расстояние*** (Manhattan distance), которое определяется как $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9.3. Малые порядковые статистики

Ранее было показано, что количество сравнений $T(n)$, которые производятся в алгоритме SELECT в наихудшем случае в ходе выбора i-й порядковой статистики из n элементов, удовлетворяет соотношению $T(n) = \Theta(n)$. Однако при этом скрытая в Θ-обозначении константа имеет довольно большую величину. Если i намного меньше n, то можно реализовать другую процедуру, в которой в качестве подпрограммы используется процедура SELECT, но в которой в наихудшем случае производится меньшее количество сравнений.

а. Опишите алгоритм, в котором для поиска среди n элементов i-го в порядке возрастания элемента требуется $U_i(n)$ сравнений, где

$$U_i(n) = \begin{cases} T(n)\,, & \text{если } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{в противном случае}\,. \end{cases}$$

(*Указание:* начните с $\lfloor n/2 \rfloor$ непересекающихся попарных сравнений и организуйте рекурсивную процедуру для множества, содержащего меньшие элементы из каждой пары.)

б. Покажите, что если $i < n/2$, то $U_i(n) = n + O(T(2i) \lg(n/i))$.

в. Покажите, что если i — константа, меньшая $n/2$, то $U_i(n) = n + O(\lg n)$.

г. Покажите, что если $i = n/k$ для $k \geq 2$, то $U_i(n) = n + O(T(2n/k) \lg k)$.

9.4. Альтернативный анализ рандомизированного выбора

В этой задаче для анализа процедуры RANDOMIZED-SELECT в том же духе, как мы проводили анализ процедуры RANDOMIZED-QUICKSORT в разделе 7.4.2, используются индикаторные случайные величины.

Как и при анализе быстрой сортировки, предполагается, что все элементы различны, и мы переименовываем элементы входного множества A как $z_1, z_2, \ldots, z_n$, где z_i представляет собой i-й в порядке возрастания элемент. Таким образом, вызов RANDOMIZED-SELECT$(A, 1, n, k)$ возвращает z_k.

Пусть для $1 \leq i < j \leq n$

$$X_{ijk} = \mathrm{I}\{\, z_i \text{ сравнивается с } z_j \text{ в некоторый момент выполнения алгоритма поиска } z_k \} \; .$$

а. Запишите точное выражение для $\mathrm{E}[X_{ijk}]$. (*Указание:* ваше выражение может иметь различные значения в зависимости от значений i, j и k.)

б. Обозначим через X_k общее количество сравнений между элементами массива A при поиске z_k. Покажите, что

$$\mathrm{E}[X_k] \leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right) \; .$$

в. Покажите, что $\mathrm{E}[X_k] \leq 4n$.

г. Сделайте вывод о том, что в предположении различности всех элементов массива A ожидаемое время работы процедуры RANDOMIZED-SELECT равно $O(n)$.

Заключительные замечания

Алгоритм поиска медиан, время работы которого в наихудшем случае линейно зависит от количества входных элементов, был разработан Блюмом (Blum), Флойдом (Floyd), Праттом (Pratt), Ривестом (Rivest) и Таржаном (Tarjan) [49]. Быстрая рандомизированная версия этого алгоритма появилась благодаря Хоару (Hoare) [168]. Флойд и Ривест [107] разработали улучшенную рандомизированную версию этого алгоритма, в которой разбиение производится относительно элемента, рекурсивно выбираемого из небольшого подмножества.

До сих пор точно неизвестно, сколько именно сравнений необходимо для поиска медианы. Нижняя граница, равная $2n$ сравнениям, была найдена Бентом (Bent) и Джоном (John) [40], а верхняя граница, равная $3n$ сравнениям, — Шёнхагом (Schönhage), Патерсоном (Paterson) и Пиппенджером (Pippenger) [300]. Дор (Dor) и Цвик (Zwick) улучшили обе эти границы. Их верхняя граница [92] немного меньше величины $2.95n$, а нижняя [93] имеет вид $(2 + \epsilon)n$, где ϵ — малая константа (что представляет собой улучшение результата Дора и др. [91]). В своей работе [270] Патерсон описал некоторые из изложенных здесь результатов, а также результаты других работ, посвященных этой теме.

9 Зак. 3726



III *Структуры данных*

Введение

Множество — это фундаментальное понятие как в математике, так и в теории вычислительных машин. Тогда как математические множества остаются неизменными, множества, которые обрабатываются в ходе выполнения алгоритмов, могут с течением времени разрастаться, уменьшаться или подвергаться другим изменениям. Назовем такие множества ***динамическими*** (dynamic). В пяти последующих главах описываются некоторые основные методы представления конечных динамических множеств и работы с ними на компьютере.

В некоторых алгоритмах, предназначенных для обработки множеств, требуется выполнять операции нескольких различных видов. Например, набор операций, используемых во многих алгоритмах, ограничивается возможностью вставлять элементы в множество, удалять их, а также проверять, принадлежит ли множеству тот или иной элемент. Динамическое множество, поддерживающее перечисленные операции, называется ***словарем*** (dictionary). В других множествах могут потребоваться более сложные операции. Например, в неубывающих очередях с приоритетами, с которыми вы ознакомились в главе 6 в контексте пирамидальной структуры данных, поддерживаются операции вставки элемента и извлечения минимального элемента. Оптимальный способ реализации динамического множества зависит от того, какие операции должны им поддерживаться.

Элементы динамического множества

В типичных реализациях динамического множества каждый его элемент представлен некоторым объектом; если в нашем распоряжении имеется указатель на объект, то можно проверять и изменять значения его атрибутов. (В разделе 10.3 обсуждается реализация объектов и указателей в средах, где они не являются базовыми типами данных.) В динамических множествах некоторых типов предполагается, что один из атрибутов объекта идентифицирует ***ключ*** (key). Если все ключи различны, то динамическое множество представимо в виде множества ключевых значений. Объекты могут содержать ***сопутствующие данные*** (satellite

data), которые находятся в других его атрибутах, но не используются реализацией множества. Кроме того, объект может содержать атрибуты, доступные для манипуляций во время выполнения операций над множеством; иногда в этих атрибутах могут храниться данные или указатели на другие объекты множества.

В некоторых динамических множествах предполагается, что их ключи являются членами полностью упорядоченного множества, например множества действительных чисел или множества всех слов, которые могут быть расположены в алфавитном порядке. Полное упорядочение, например, позволяет определить минимальный элемент множества или говорить о ближайшем элементе множества, превышающем заданный.

Операции над динамическими множествами

Операции над динамическим множеством можно разбить на две категории: ***запросы*** (queries), которые просто возвращают информацию о множестве, и ***модифицирующие операции*** (modifying operations), изменяющие множество. Ниже приведен список типичных операций. В каждом конкретном приложении требуется, чтобы были реализованы лишь некоторые из них.

SEARCH(S, k)
: Запрос, который возвращает указатель на элемент x заданного множества S, для которого $x.key = k$, или значение NIL, если в множестве S такой элемент отсутствует.

INSERT(S, x)
: Модифицирующая операция, которая пополняет заданное множество S одним элементом, на который указывает x. Обычно предполагается, что выполнена предварительная инициализация всех атрибутов элемента x, требующихся реализации множества.

DELETE(S, x)
: Модифицирующая операция, удаляющая из заданного множества S элемент, на который указывает x. (Обратите внимание, что в этой операции используется указатель на элемент, а не его ключевое значение.)

MINIMUM(S)
: Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наименьшим ключом.

MAXIMUM(S)
: Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наибольшим ключом.

SUCCESSOR(S, x)
: Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим соседом ключа элемента x и превышает его. Если же x — максимальный элемент множества S, то возвращается значение NIL.

PREDECESSOR(S, x)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим меньшим по значению соседом ключа элемента x. Если же x — минимальный элемент множества S, то возвращается значение NIL.

Запросы SUCCESSOR и PREDECESSOR в некоторых ситуациях обобщаются на множества, в которых не все ключи различаются. Для множества, состоящего из n элементов, обычно принимается допущение, что вызов операции MINIMUM, после которой $n - 1$ раз вызывается операция SUCCESSOR, позволяет пронумеровать элементы множества в порядке сортировки.

Время, необходимое для выполнения операций множества, обычно измеряется в единицах, связанных с размером множества, который указывается в качестве одного из аргументов. Например, в главе 13 описывается структура данных, способная поддерживать все перечисленные выше операции, причем время их выполнения на множестве размером n выражается как $O(\lg n)$.

Обзор части III

В главах 10–14 описываются несколько структур данных, с помощью которых можно реализовать динамические множества. Многие из этих структур данных будут использоваться впоследствии при разработке эффективных алгоритмов, позволяющих решать разнообразные задачи. Еще одна важная структура данных, пирамида, уже рассматривалась в главе 6.

В главе 10 представлены основные приемы работы с такими простыми структурами данных, как стеки, очереди, связанные списки и корневые деревья. В ней также показано, как можно реализовать в средах программирования объекты и указатели, в которых они не поддерживаются в качестве примитивов. Бóльшая часть материала этой главы должна быть знакома всем, кто освоил вводный курс программирования.

Глава 11 знакомит читателя с хеш-таблицами, поддерживающими такие словарные операции, как INSERT, DELETE и SEARCH. В наихудшем случае операция поиска в хеш-таблицах выполняется в течение времени $\Theta(n)$, однако математическое ожидание времени выполнения подобных операций равно $O(1)$. Анализ хеширования основывается на теории вероятности, однако для понимания большей части материала этой главы не требуются предварительные знания в этой области.

Описанные в главе 12 бинарные деревья поиска поддерживают все перечисленные выше операции динамических множеств. В наихудшем случае для выполнения каждой такой операции на n-элементном множестве требуется время $\Theta(n)$, однако при случайном построении бинарных деревьев поиска математическое ожидание времени работы каждой операции равно $O(\lg n)$. Бинарные деревья поиска служат основой для многих других структур данных.

С красно-черными деревьями, представляющими собой разновидность бинарных деревьев поиска, вы познакомитесь в главе 13. В отличие от обычных бинарных деревьев поиска, красно-черные деревья всегда работают хорошо: в наи-

худшем случае операции над ними выполняются за время $O(\lg n)$. Красно-черное дерево представляет собой сбалансированное дерево поиска; в главе 18 части V представлено сбалансированное дерево поиска другого вида, получившее название "В-дерево". Несмотря на то что механизмы работы красно-черных деревьев несколько запутаны, из этой главы можно получить детальное представление об их свойствах без подробного изучения этих механизмов. Тем не менее будет довольно поучительно, если вы внимательно ознакомитесь со всем представленным в главе материалом.

В главе 14 показано, как расширить красно-черные деревья для поддержки операций, отличных от перечисленных выше базовых. Сначала будет рассмотрено расширение красно-черных деревьев, обеспечивающее динамическую поддержку порядковых статистик для множества ключей, а затем — для поддержки интервалов действительных чисел.

Глава 10. Элементарные структуры данных

В этой главе рассматривается представление динамических множеств простыми структурами данных, в которых используются указатели. Несмотря на то что с помощью указателей можно сформировать многие сложные структуры данных, здесь будут представлены лишь простейшие из них: стеки, очереди, связанные списки и деревья. Мы также рассмотрим метод, позволяющий синтезировать из массивов объекты и указатели.

10.1. Стеки и очереди

Стеки и очереди представляют собой динамические множества, элементы из которых удаляются с помощью предварительно определенной операции DELETE. Первым из ***стека*** (stack) удаляется элемент, который был помещен туда последним: в стеке реализуется стратегия ***"последним вошел — первым вышел"*** (last-in, first-out — LIFO). Аналогично в ***очереди*** (queue) всегда удаляется элемент, который содержится в множестве дольше других: в очереди реализуется стратегия ***"первым вошел — первым вышел"*** (first-in, first-out — FIFO). Существует несколько эффективных способов реализации стеков и очередей в компьютере. В данном разделе будет показано, как реализовать обе эти структуры данных с помощью обычного массива.

Стеки

Операция вставки INSERT применительно к стекам часто называется записью в стек PUSH, а операция удаления DELETE, которая вызывается без передачи аргумента, — снятием со стека POP.

Как видно из рис. 10.1, стек, способный вместить не более n элементов, можно реализовать с помощью массива $S[1..n]$. Этот массив обладает атрибутом $S.top$, представляющим собой индекс последнего помещенного в стек элемента. Стек состоит из элементов $S[1..S.top]$, где $S[1]$ — элемент на дне стека, а $S[S.top]$ — элемент на его вершине.

Если $S.top = 0$, то стек не содержит ни одного элемента и является ***пустым*** (empty). Протестировать стек на наличие в нем элементов можно с помощью опе-

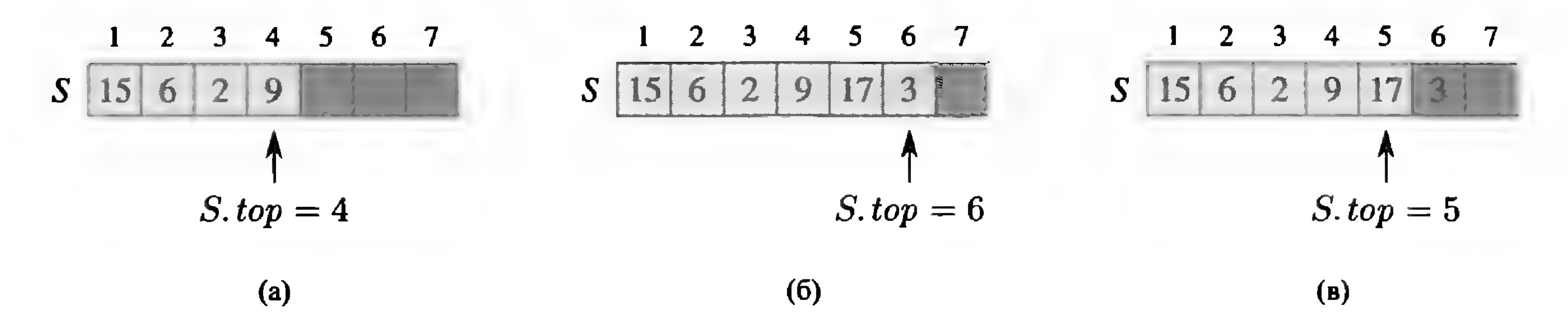


Рис. 10.1. Реализация стека S с помощью массива. Элементы стека находятся только в тех позициях массива, которые отмечены светло-серым цветом. **(а)** Стек S состоит из четырех элементов. На вершине стека находится элемент 9. **(б)** Стек S после вызовов PUSH(S, 17) и PUSH(S, 3). **(в)** Стек S после вызова процедуры POP(S), которая возвращает помещенное в стек последним значение 3. Несмотря на то что элемент 3 все еще находится в массиве, он больше не принадлежит стеку; теперь на вершине стека располагается элемент 17.

рации-запроса STACK-EMPTY. Если выполняется попытка снятия элемента с пустого стека, говорят, что он ***опустошается*** (underflow), что обычно приводит к ошибке. Если значение $S.top$ больше n, то стек ***переполняется*** (overflow). (В представленном ниже псевдокоде возможное переполнение во внимание не принимается.)

Каждую операцию над стеком можно легко реализовать несколькими строками кода.

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE
```

```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

На рис. 10.1 показано воздействие на стек модифицирующих операций PUSH и POP. Каждая из указанных трех операций над стеком выполняется за время $O(1)$.

Очереди

Применительно к очередям операция вставки называется ENQUEUE (поместить в очередь), а операция удаления — DEQUEUE (вывести из очереди). Подобно стековой операции POP, операция DEQUEUE не требует передачи элемента массива в виде аргумента. Благодаря свойству FIFO очередь подобна, например, живой

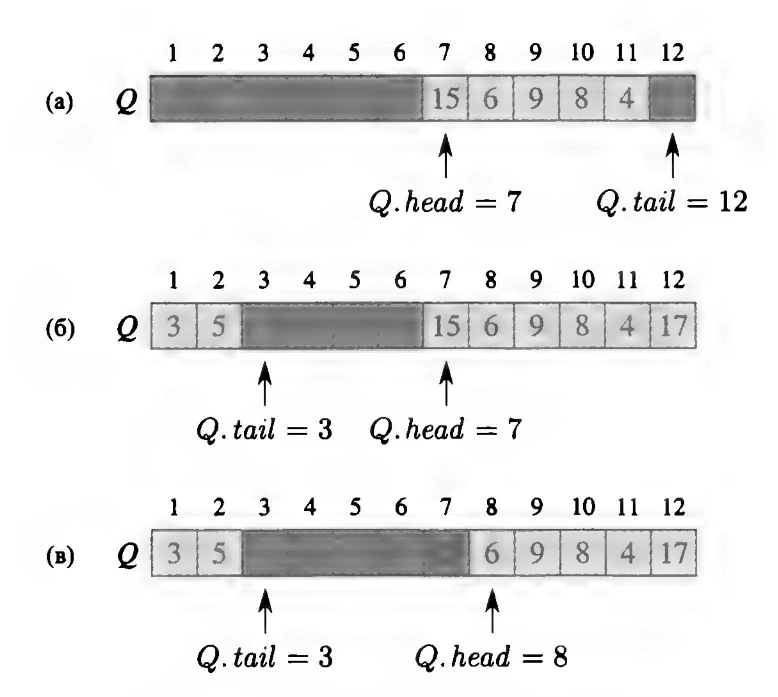


Рис. 10.2. Реализация очереди с помощью массива $Q[1\,..\,12]$. Элементы очереди содержатся только в светло-серых ячейках. **(а)** Очередь содержит пять элементов в позициях $Q[7\,..\,11]$. **(б)** Конфигурация очереди после вызовов ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$ и ENQUEUE$(Q, 5)$. **(в)** Конфигурация очереди после того, как вызов DEQUEUE(Q) вернул ключевое значение 15, ранее находившееся в голове очереди. Новый элемент в голове очереди имеет ключ 6.

очереди к врачу в поликлинике. У нее имеются ***голова*** (head) и ***хвост*** (tail). Когда элемент помещается в очередь, он занимает место в ее хвосте, точно так же, как человек занимает очередь последним, чтобы попасть на прием к врачу. Из очереди всегда выводится элемент, который находится в ее головной части аналогично тому, как в кабинет врача всегда заходит больной, который ждал дольше всех.

На рис. 10.2 показан один из способов, который позволяет с помощью массива $Q[1\,..\,n]$ реализовать очередь, состоящую не более чем из $n - 1$ элементов. Эта очередь обладает атрибутом $Q.head$, который является индексом головного элемента или указателем на него; атрибут $Q.tail$ индексирует местоположение, куда будет добавляться новый элемент. Элементы очереди расположены в ячейках $Q.head, Q.head + 1, \ldots, Q.tail - 1$, которые циклически замкнуты в том смысле, что ячейка 1 следует сразу же после ячейки n в циклическом порядке. При выполнении условия $Q.head = Q.tail$ очередь пуста. Изначально выполняется соотношение $Q.head = Q.tail = 1$. Если очередь пуста, то при попытке удалить из нее элемент происходит ошибка опустошения. Если $Q.head = Q.tail + 1$, или $Q.head = 1$ и $Q.tail = Q.length$, то очередь заполнена, и попытка добавить в нее элемент приводит к ее переполнению.

В наших процедурах ENQUEUE и DEQUEUE проверка ошибок опустошения и переполнения не проводится. (В упр. 10.1.4 предлагается добавить в процедуры соответствующий код.) В псевдокоде предполагается, что $n = Q.length$.

```
ENQUEUE(Q, x)
Q[Q.tail] = x
if Q.tail == Q.length
    Q.tail = 1
else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
    Q.head = 1
else Q.head = Q.head + 1
return x
```

На рис. 10.2 показана работа процедур ENQUEUE и DEQUEUE. Каждая операция выполняется за время $O(1)$.

Упражнения

10.1.1

Используя рис. 10.1 в качестве образца, проиллюстрируйте результат воздействия на изначально пустой стек S, хранящийся в массиве $S[1..6]$, последовательности операций PUSH$(S,4)$, PUSH$(S,1)$, PUSH$(S,3)$, POP(S), PUSH$(S,8)$ и POP(S).

10.1.2

Объясните, как с помощью одного массива $A[1..n]$ можно реализовать два стека таким образом, чтобы ни один из них не переполнялся, пока суммарное количество элементов в обоих стеках не достигнет n. Операции PUSH и POP должны выполняться за время $O(1)$.

10.1.3

Используя рис. 10.2 в качестве образца, проиллюстрируйте результат каждой из операций в последовательности ENQUEUE$(Q,4)$, ENQUEUE$(Q,1)$, ENQUEUE$(Q,3)$, DEQUEUE(Q), ENQUEUE$(Q,8)$ и DEQUEUE(Q) над изначально пустой очередью Q, хранящейся в массиве $Q[1..6]$.

10.1.4

Перепишите процедуры ENQUEUE и DEQUEUE так, чтобы они корректно обнаруживали опустошение и переполнение очереди.

10.1.5

При работе со стеком элементы можно добавлять в него и извлекать из него только с одного конца. Очередь позволяет добавлять элементы с одного конца, а извлекать — с другого. ***Очередь с двусторонним доступом***, или ***дек*** (deque), предоставляет возможность производить вставку и удаление с обоих концов. Напишите четыре процедуры, выполняющиеся в течение времени $O(1)$ и позволяющие вставлять и удалять элементы с обоих концов дека, реализованного с помощью массива.

10.1.6

Покажите, как реализовать очередь с помощью двух стеков. Проанализируйте время работы операций, которые выполняются с ее элементами.

10.1.7

Покажите, как реализовать стек с помощью двух очередей. Проанализируйте время работы операций, которые выполняются с его элементами.

10.2. Связанные списки

Связанный список (linked list) — это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. Связанные списки обеспечивают простое и гибкое представление динамических множеств и поддерживают все операции (хотя и не всегда достаточно эффективно), перечисленные на с. 261.

Как показано на рис. 10.3, каждый элемент ***дважды связанного списка*** (doubly linked list) L — это объект с одним атрибутом *key* и двумя атрибутами-указателями: *next* (следующий) и *prev* (предыдущий). Этот объект может также содержать другие сопутствующие данные. Для заданного элемента списка x указатель $x.next$ указывает на следующий элемент связанного списка, а указатель $x.prev$ — на предыдущий. Если $x.prev = \text{NIL}$, у элемента x нет предшественника, и, следовательно, он является первым, т.е. ***головным*** в списке. Если $x.next = \text{NIL}$, то у элемента x нет последующего, а значит, он является последним, т.е. ***хвостовым*** в списке. Атрибут $L.head$ указывает на первый элемент списка. Если $L.head = \text{NIL}$, то список пуст.

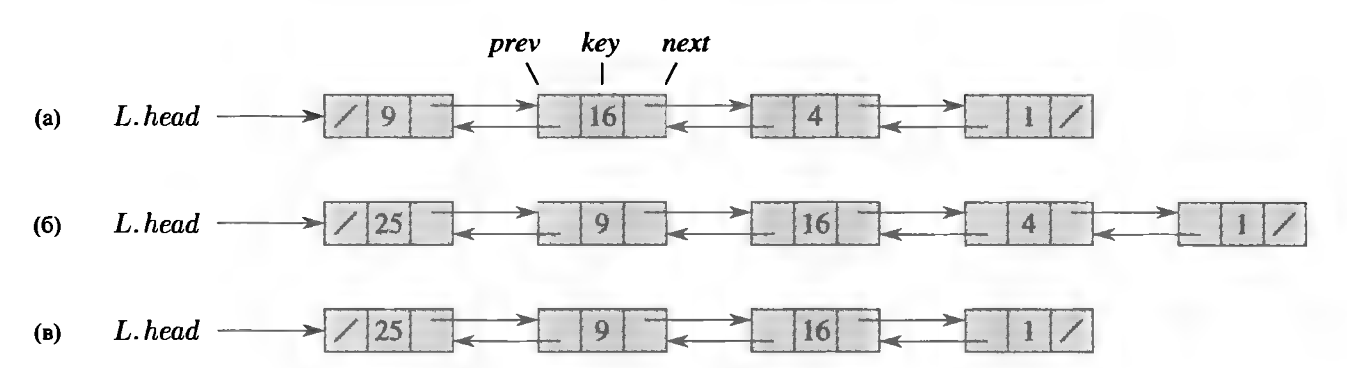


Рис. 10.3. (а) Дважды связанный список L, представляющий динамическое множество $\{1, 4, 9, 16\}$. Каждый элемент списка является объектом с атрибутами, представляющими ключ и указатели (показанные стрелками) на следующий и предыдущий объекты. Атрибут *next* хвостового и атрибут *prev* головного элементов равны NIL, что показано диагональной косой чертой. Атрибут $L.head$ указывает на голову списка. **(б)** После выполнения LIST-INSERT(L, x), где $x.key = 25$, связанный список в качестве головного элемента получает новый объект с ключом 25. Этот новый объект указывает на старый головной элемент с ключом 9. **(в)** Результат последующего вызова LIST-DELETE(L, x), где x указывает на объект с ключом 4.

Списки могут быть разных видов. Список может быть однократно или дважды связанным, отсортированным или неотсортированным, кольцевым или некольцевым. Если список ***однократно связанный (однонаправленный)*** (singly linked), то указатель *prev* в его элементах отсутствует. Если список ***отсортирован*** (sorted), то его линейный порядок соответствует линейному порядку его ключей; в этом случае минимальный элемент находится в голове списка, а максимальный — в его хвосте. Если же список не отсортирован, то его элементы могут располагаться в произвольном порядке. Если список ***кольцевой*** (circular list), то указатель *prev* его головного элемента указывает на его хвост, а указатель *next* хвостового элемента — на головной элемент. Такой список можно рассматривать как замкнутый в виде кольца набор элементов. В оставшейся части раздела предполагается, что списки, с которыми нам придется работать, — неотсортированные дважды связанные.

Поиск в связанном списке

Вызов LIST-SEARCH(L, k) находит в списке L первый элемент с ключом k путем простого линейного поиска и возвращает указатель на найденный элемент. Если элемент с ключом k в списке отсутствует, возвращается значение NIL. Процедура LIST-SEARCH$(L, 4)$, вызванная для связанного списка, изображенного на рис. 10.3, (а), возвращает указатель на третий элемент, а вызов LIST-SEARCH$(L, 7)$ — значение NIL.

LIST-SEARCH(L, k)

$x = L.head$
while $x \neq$ NIL и $x.key \neq k$
    $x = x.next$
return x

Поиск с помощью процедуры LIST-SEARCH в списке, состоящем из n элементов, в наихудшем случае выполняется за время $\Theta(n)$, поскольку может понадобиться просмотреть весь список.

Вставка в связанный список

Если имеется элемент x, атрибут *key* которого предварительно установлен, то процедура LIST-INSERT вставляет элемент x в начало списка (рис. 10.3, (б)).

LIST-INSERT(L, x)

$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL

(Вспомните, что наша запись атрибутов допускает каскадирование, так что *L.head.prev* означает атрибут *prev* объекта, на который указывает атрибут *L.head*.) Время работы LIST-INSERT со списком из n элементов равно $O(1)$.

Удаление из связанного списка

Представленная ниже процедура LIST-DELETE удаляет элемент x из связанного списка L. В процедуру необходимо передать указатель на элемент x, после чего она удаляет x из списка путем обновления указателей. Чтобы удалить элемент с заданным ключом, необходимо сначала вызвать процедуру LIST-SEARCH для получения указателя на элемент.

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

На рис. 10.3, (в) показано, как элемент удаляется из связанного списка. Время работы процедуры LIST-DELETE равно $O(1)$, но если требуется удалить элемент с заданным ключом, то в наихудшем случае требуется время $\Theta(n)$, поскольку сначала нужно вызвать процедуру LIST-SEARCH для поиска удаляемого элемента.

Ограничители

Код процедуры LIST-DELETE мог бы быть проще, если бы можно было игнорировать граничные условия в голове и хвосте списка.

```
LIST-DELETE'(L, x)
x.prev.next = x.next
x.next.prev = x.prev
```

Ограничитель(sentinel) — это фиктивный объект, упрощающий учет граничных условий. Например, предположим, что в списке L предусмотрен объект *L.nil*, представляющий значение NIL, но при этом содержащий все атрибуты, которые имеются у других элементов. Когда в коде происходит обращение к значению NIL, оно заменяется обращением к ограничителю *L.nil*. Как показано на рис. 10.4, наличие ограничителя преобразует обычный дважды связанный список в ***циклический дважды связанный список с ограничителем***. В таком списке ограничитель *L.nil* расположен между головой и хвостом. Атрибут *L.nil.next* указывает на голову списка, а атрибут *L.nil.prev* — на его хвост. Аналогично атрибуты *next* хвостового элемента и *prev* головного элемента указывают на элемент *L.nil*. Поскольку атрибут *L.nil.next* указывает на голову списка, можно упразднить атрибут *L.head*, заменив ссылки на него ссылками на *L.nil.next*. Как видно на рис. 10.4, (а), пустой список содержит только ограничитель, и как *L.nil.next*, так и *L.nil.prev* указывают на *L.nil*.

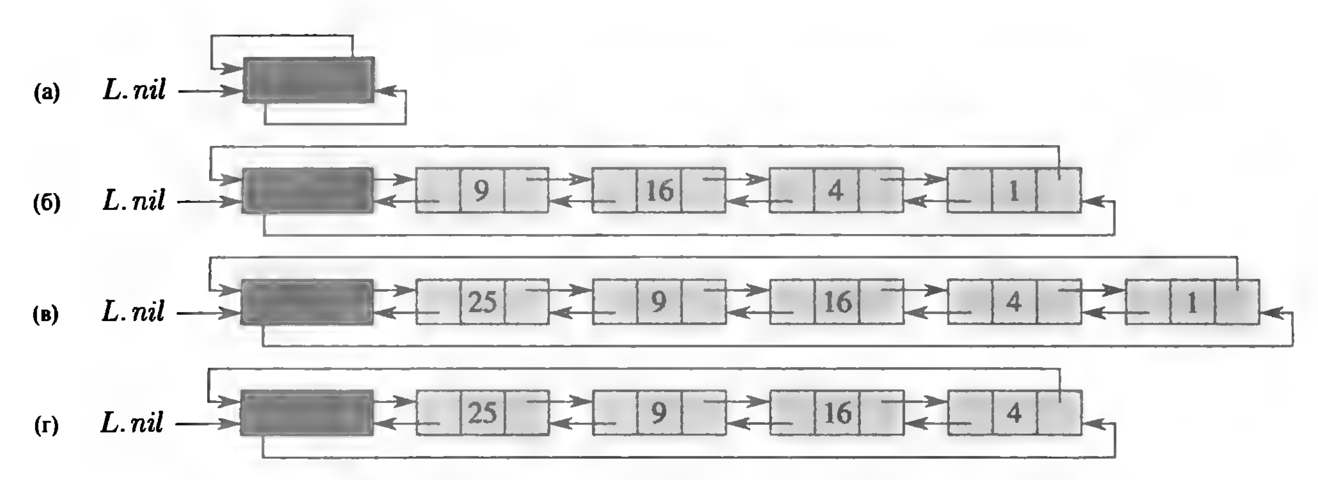


Рис. 10.4. Циклический дважды связанный список с ограничителями. Ограничитель $L.nil$ находится между головой и хвостом. Атрибут $L.head$ больше не нужен, поскольку доступ к голове списка получается с помощью $L.nil.next$. **(а)** Пустой список. **(б)** Связанный список с рис. 10.3, (а), с ключом 9 в голове и ключом 1 в хвосте списка. **(в)** Список после выполнения LIST-INSERT$'(L, x)$, где $x.key = 25$. Новый объект становится головой списка. **(г)** Список после удаления объекта с ключом 1. Новым хвостом становится объект с ключом 4.

Код LIST-SEARCH остается прежним, но обращения к NIL и $L.head$ изменены, как указано выше.

```
LIST-SEARCH'(L, k)
x = L.nil.next
while x ≠ L.nil и x.key ≠ k
    x = x.next
return x
```

Удаление элемента из списка производится с помощью уже описанной двустрочной процедуры LIST-DELETE$'$. Для вставки элемента в список используется приведенная ниже процедура.

```
LIST-INSERT'(L, x)
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

Действие процедур LIST-INSERT$'$ и LIST-DELETE$'$ на список-образец показано на рис. 10.4.

Применение ограничителей редко приводит к улучшению асимптотического времени обработки структур данных, однако может уменьшить величину постоянных множителей. Благодаря использованию ограничителей в циклах обычно не столько увеличивается скорость работы кода, сколько повышается его ясность. Например, представленный выше код, предназначенный для обработки связанного списка, упрощается в результате применения ограничителей, но сэкономленное в процедурах LIST-INSERT$'$ и LIST-DELETE$'$ время составляет всего лишь

$O(1)$. Однако в других ситуациях применение ограничителей позволяет сделать код в циклах компактнее, а иногда даже уменьшить время работы в n или n^2 раз.

Ограничители не стоит применять необдуманно. Если в программе обрабатывается большое количество маленьких списков, то дополнительное пространство, занимаемое ограничителями, может стать причиной значительного перерасхода памяти. В этой книге ограничители применяются лишь тогда, когда они действительно упрощают код.

Упражнения

10.2.1

Можно ли реализовать операцию INSERT над динамическим множеством в однократно связанном списке так, чтобы время ее работы было равно $O(1)$? А операцию DELETE?

10.2.2

Реализуйте стек с помощью однократно связанного списка L. Операции PUSH и POP должны по-прежнему выполняться за время $O(1)$.

10.2.3

Реализуйте очередь с помощью однократно связанного списка L. Операции ENQUEUE и DEQUEUE должны по-прежнему выполняться за время $O(1)$.

10.2.4

Как уже говорилось, в каждой итерации цикла, входящего в состав процедуры LIST-SEARCH$'$, необходимо выполнить две проверки: $x \neq L.nil$ и $x.key \neq k$. Покажите, как избежать проверки $x \neq L.nil$ в каждой итерации.

10.2.5

Реализуйте базовые словарные операции INSERT, DELETE и SEARCH с помощью однократно связанного циклического списка. Определите время работы этих процедур.

10.2.6

Операция UNION (объединение) над динамическим множеством принимает в качестве входных данных два непересекающихся множества S_1 и S_2 и возвращает множество $S = S_1 \cup S_2$, состоящее из всех элементов множеств S_1 и S_2. В результате выполнения этой операции множества S_1 и S_2 обычно разрушаются. Покажите, как организовать поддержку операции UNION со временем работы $O(1)$ с помощью подходящей списочной структуры данных.

10.2.7

Разработайте нерекурсивную процедуру со временем работы $\Theta(n)$, обращающую порядок расположения элементов в однократно связанном списке. Процедура должна использовать некоторый постоянный объем памяти, помимо памяти, необходимой для хранения самого списка.

10.2.8 ★

Объясните, как можно реализовать дважды связанный список, используя при этом всего лишь один указатель $x.np$ в каждом элементе вместо обычных двух (*next* и *prev*). Предполагается, что значения всех указателей могут рассматриваться как k-битовые целые числа, а величина $x.np$ определяется как $x.np = x.next$ XOR $x.prev$, т.е. как k-битовое "исключающее или" значений $x.next$ и $x.prev$. (Значение NIL представляется нулем.) Не забудьте указать, какая информация нужна для доступа к голове списка. Покажите, как реализовать операции SEARCH, INSERT и DELETE в таком списке. Покажите также, как можно обратить порядок элементов в таком списке за время $O(1)$.

10.3. Реализация указателей и объектов

Как реализовать указатели и объекты в языках, где их просто нет? В данном разделе мы ознакомимся с двумя путями реализации связанных структур данных, в которых такой тип данных, как указатель, в явном виде не используется. Объекты и указатели будут созданы на основе массивов и индексов.

Представление объектов с помощью нескольких массивов

Набор объектов с одинаковыми атрибутами можно представить с помощью массивов. В качестве примера рассмотрим рис. 10.5, на котором проиллюстрирована реализация с помощью трех массивов связанного списка, представленного на рис. 10.3, (а). В массиве *key* содержатся значения ключей элементов, входящих в данный момент в динамическое множество, а указатели хранятся в массивах *next* и *prev*. Для заданного индекса массива x элементы $key[x]$, $next[x]$ и $prev[x]$ представляют объект в связанном списке. В такой интерпретации указатель x — это просто общий индекс в массивах *key*, *next* и *prev*.

На рис. 10.3, (а) объект с ключом 4 следует в связанном списке за объектом с ключом 16. На рис. 10.5 ключ 4 располагается в $key[2]$, а ключ 16 — в $key[5]$, так что $next[5] = 2$ и $prev[2] = 5$. Хотя в атрибуте *next* хвоста и *prev* головы

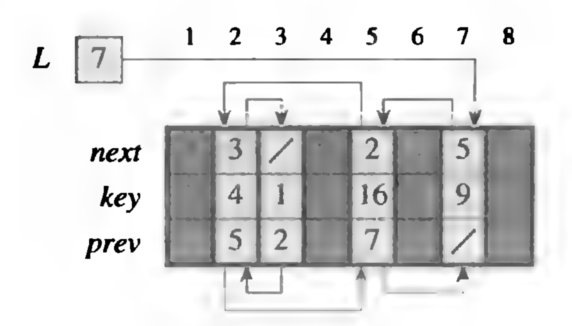


Рис. 10.5. Связанный список с рис. 10.3, (а), представленный массивами *key*, *next* и *prev*. Каждый вертикальный срез массивов представляет один объект. Сохраненные указатели соответствуют индексам массива, показанным в верхней части рисунка; стрелки указывают, как они должны быть интерпретированы. Светлая штриховка указывает позиции, содержащие элементы списка. Переменная L хранит индекс головы списка.

списка показана константа NIL, обычно для этой цели используется целое число (такое, как 0 или -1), которое не может представлять реальный индекс массива. Переменная L хранит индекс головы списка.

Представление объектов с помощью одного массива

Обычно слова в памяти компьютера адресуются с помощью целых чисел от 0 до $M-1$, где M — это достаточно большое целое число. Во многих языках программирования объект занимает непрерывную область памяти компьютера, а указатель представляет собой просто адрес первой ячейки памяти, где находится начало объекта. Другие ячейки памяти, занимаемые объектом, можно индексировать путем добавления к указателю величины соответствующего смещения.

Этой же стратегией можно воспользоваться при реализации объектов в средах программирования, в которых отсутствуют указатели. Например, на рис. 10.6 показано, как можно реализовать связанный список, знакомый нам из рис. 10.3, (а) и 10.5, с помощью одного массива A. Объект занимает подмассив смежных элементов $A[j \,..\, k]$. Каждый атрибут объекта соответствует смещению, величина которого принимает значения от 0 до $k-j$, а указателем на объект является индекс j. Каждый элемент списка — это объект, занимающий по три расположенных рядом элемента массива. Указатель на объект — это индекс его первого элемента. Объекты, в которых содержатся элементы списка, на рисунке отмечены светло-серым цветом. На рис. 10.6 смещения, отвечающие атрибутам *key*, *next* и *prev*, равны 0, 1 и 2 соответственно. Чтобы прочесть значение атрибута $i.prev$ для данного указателя i, к значению указателя добавляется величина смещения 2, в результате чего получается $A[i+2]$.

Представление в виде единственного массива можно считать более гибким в том плане, что с его помощью в одном и том же массиве можно хранить объекты различной длины. Задача управления такими неоднородными наборами объектов сложнее, чем аналогичная задача для однородного набора объектов, где все объекты имеют одинаковые атрибуты. Поскольку большинство структур данных, которое нам предстоит рассмотреть, состоит из однородных элементов, для наших целей достаточно использовать представление объектов с помощью нескольких массивов.

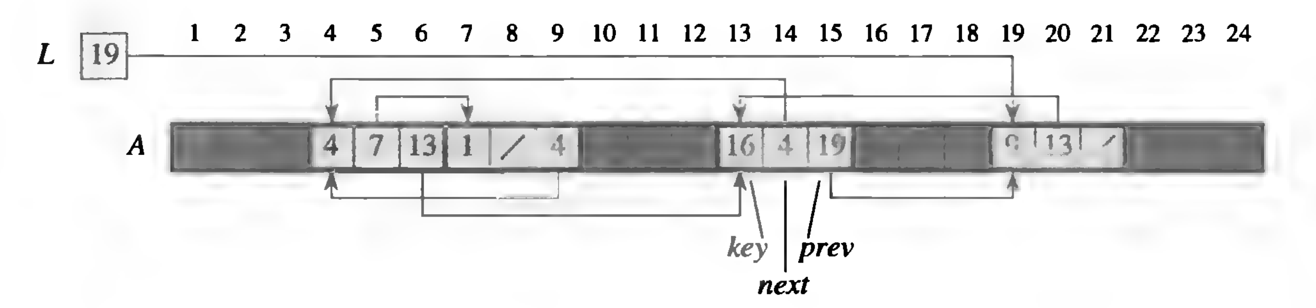


Рис. 10.6. Связанный список с рис. 10.3, (а) и 10.5, представленный единственным массивом A. Каждый элемент списка является объектом, занимающим подмассив из трех смежных ячеек массива. Атрибуты *key*, *next* и *prev* в каждом объекте отвечают смещениям 0, 1 и 2 соответственно. Указатель на объект представляет собой индекс первого элемента объекта. Объекты, содержащие элементы списка, выделены светлой штриховкой; стрелки показывают порядок в списке.

Выделение и освобождение объектов

При добавлении нового элемента в динамическое множество, представленное дважды связанным списком, необходимо получить указатель на не используемый в настоящее время объект в представлении связанного списка. Таким образом, было бы полезно управлять хранением объектов, не используемых в данный момент в представлении связного списка, чтобы можно было легко их получать при необходимости. В некоторых системах за выявление неиспользуемых объектов отвечают ***сборщики мусора*** (garbage collector). Однако многие приложения достаточно просты и способны сами отвечать за возврат неиспользуемых объектов модулю управления памятью. Давайте рассмотрим задачу выделения и освобождения однородных объектов на примере дважды связанного списка, представленного с помощью нескольких массивов.

Предположим, что в таком представлении используются массивы длиной m и что в какой-то момент динамическое множество содержит $n \leq m$ элементов. В этом случае n объектов представляют элементы, которые находятся в данный момент в динамическом множестве, а $m - n$ объектов ***свободны***. Их можно использовать для представления элементов, которые будут вставляться в динамическое множество в будущем.

Свободные объекты хранятся в однократно связанном списке, который мы назовем ***списком свободных позиций*** (free list). Список свободных позиций использует только массив $next$, в котором хранятся указатели $next$ списка. Голова списка свободных позиций находится в глобальной переменной *free*. Если динамическое множество, представленное связанным списком L, не является пустым, список свободных позиций оказывается "переплетенным" со списком L, как показано на рис. 10.7. Заметим, что каждый объект в таком представлении находится либо в списке L, либо в списке свободных позиций, но не в обоих списках одновременно.

Список свободных позиций работает как стек: очередной выделяемый объект является последним освобожденным. Таким образом, реализовать процедуры выделения и освобождения памяти для объектов можно с помощью стековых операций PUSH и POP. Глобальная переменная *free*, использующаяся в приведенных ниже процедурах, указывает на первый элемент списка свободных позиций.

```
ALLOCATE-OBJECT()
if free == NIL
    error "Не хватает памяти"
else x = free
    free = x.next
    return x
```

```
FREE-OBJECT(x)
x.next = free
free = x
```

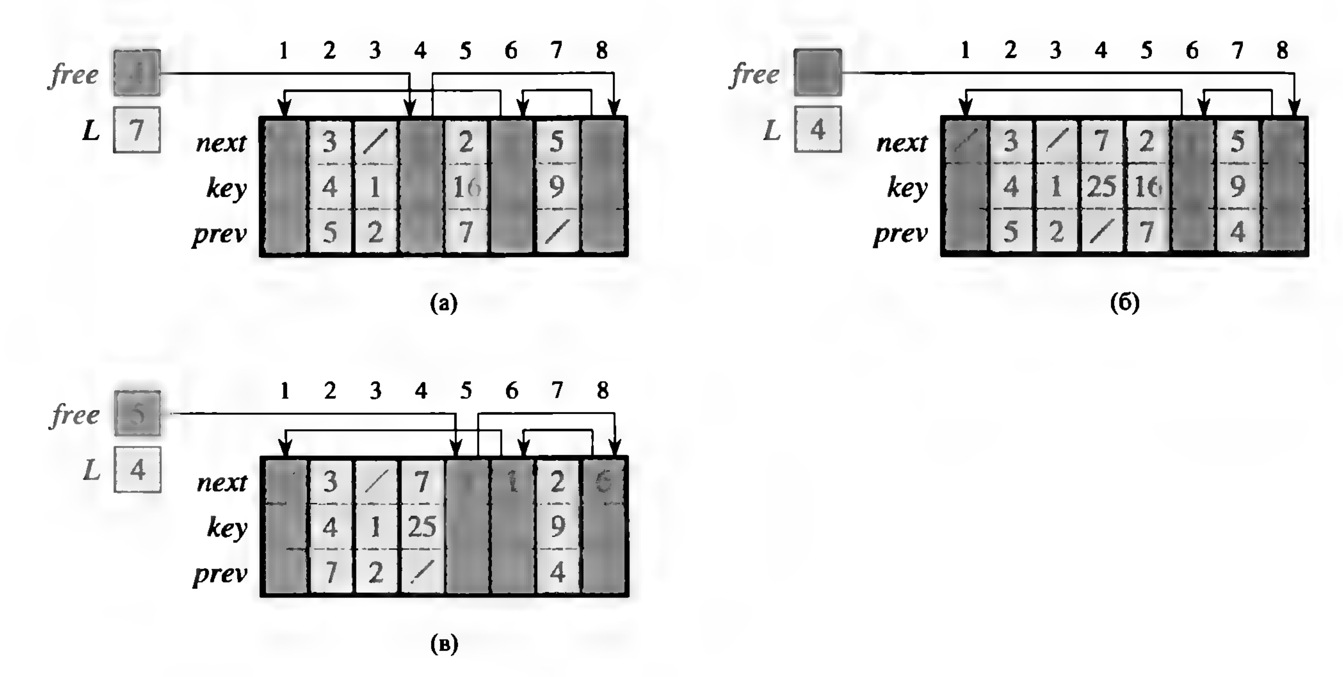


Рис. 10.7. Результат вызовов процедур ALLOCATE-OBJECT и FREE-OBJECT. **(а)** Список с рис. 10.5 (светлая штриховка) и список свободных позиций (темная штриховка). Стрелки показывают структуру списка свободных позиций. **(б)** Результат вызова ALLOCATE-OBJECT() (возвращающего индекс 4), установки $key[4]$ равным 25 и вызова LIST-INSERT$(L, 4)$. Головным элементом нового списка свободных позиций является объект 8, который в списке свободных позиций представлял собой $next[4]$. **(в)** После выполнения LIST-DELETE$(L, 5)$ вызывается FREE-OBJECT(5). Объект 5 становится новым заголовком списка свободных позиций; следующим в списке за ним идет объект 8.

Изначально список свободных позиций содержит все n объектов, для которых не выделено место. Когда в списке свободных позиций больше не остается элементов, процедура ALLOCATE-OBJECT выдает сообщение об ошибке. Один список свободных позиций может обслуживать несколько связанных списков. Но рис. 10.8 показаны два связанных списка и список свободных позиций, связанные посредством массивов key, $next$ и $prev$.

Время выполнения обеих процедур равно $O(1)$, что делает их весьма практичными. Эти процедуры можно модифицировать так, чтобы они работали с любым набором однородных объектов, лишь бы один из атрибутов объекта работал в качестве атрибута $next$ списка свободных позиций.



Рис. 10.8. Связанные списки L_1 (светло-серый) и L_2 (темно-серый) и обслуживающий их список свободных позиций (черный).

Упражнения

10.3.1

Изобразите последовательность $\langle 13, 4, 8, 19, 5, 11 \rangle$, хранящуюся в дважды связанном списке, представленном с помощью нескольких массивов. Выполните это же задание для представления с помощью одного массива.

10.3.2

Разработайте процедуры ALLOCATE-OBJECT и FREE-OBJECT для однородного набора объектов, реализованного с помощью одного массива.

10.3.3

Почему нет необходимости присваивать или вновь устанавливать значения атрибутов *prev* при реализации процедур ALLOCATE-OBJECT и FREE-OBJECT?

10.3.4

Зачастую (например, при страничной организации виртуальной памяти) все элементы списка желательно располагать компактно, в непрерывном участке памяти, например, используя m первых позиций в представлении списка несколькими массивами. Поясните, как реализовать процедуры ALLOCATE-OBJECT и FREE-OBJECT так, чтобы получить компактное представление. Считаем, что нет никаких указателей на элементы связанного списка извне. (*Указание:* воспользуйтесь реализацией стека с помощью массива.)

10.3.5

Пусть L — дважды связанный список длиной n, который хранится в массивах *key*, *prev* и *next* длиной m. Предположим, что управление этими массивами осуществляется с помощью процедур ALLOCATE-OBJECT и FREE-OBJECT, использующих дважды связанный список свободных позиций F. Предположим также, что ровно n из m элементов находятся в списке L, а остальные $m - n$ элементов — в списке свободных позиций. Разработайте процедуру COMPACTIFY-LIST(L, F), которая в качестве параметров получает список L и список свободных позиций F и перемещает элементы списка L таким образом, чтобы они занимали ячейки массива с индексами $1, 2, \ldots, n$, а также преобразует список свободных позиций F так, что он остается корректным и содержит ячейки массива с индексами $n + 1, n + 2, \ldots, m$. Время работы этой процедуры должно быть равным $\Theta(n)$, а объем используемой ею дополнительной памяти не должен превышать некоторую фиксированную величину. Докажите корректность разработанной процедуры.

10.4. Представление корневых деревьев

Приведенные в предыдущем разделе методы представления списков подходят для любых однородных структур данных. Данный раздел посвящен задаче пред-

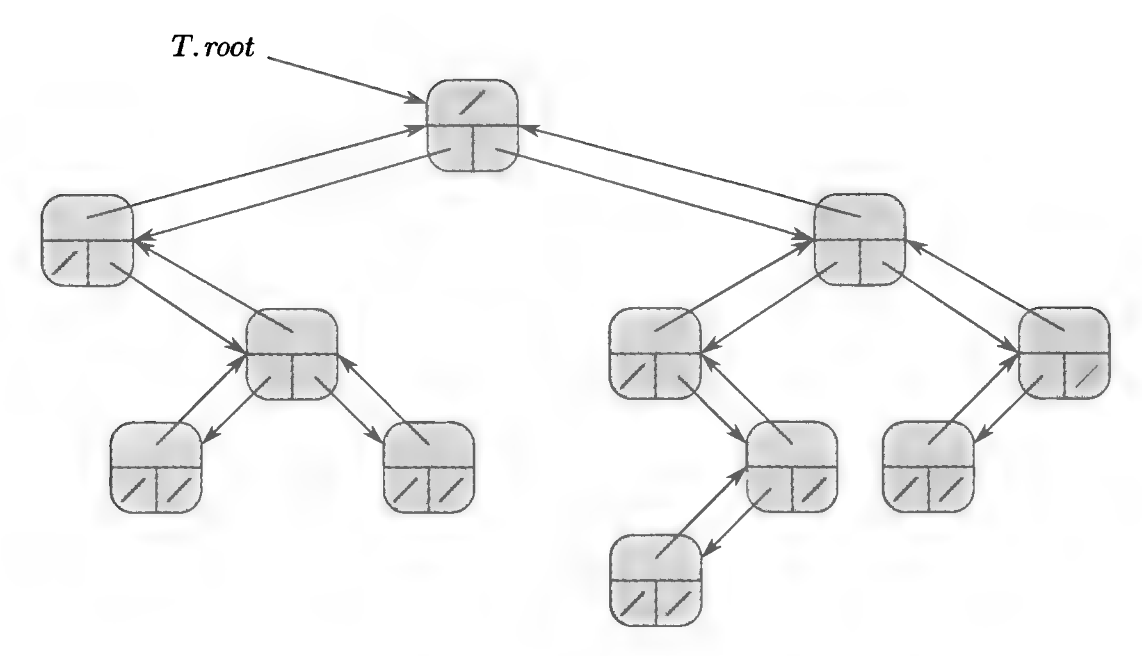


Рис. 10.9. Представление бинарного дерева T. Каждый узел x имеет атрибуты $x.p$ (вверх), $x.left$ (влево вниз) и $x.right$ (вправо вниз). Атрибуты key не показаны.

ставления корневых деревьев с помощью связанных структур данных. Мы начнем с рассмотрения бинарных деревьев, а затем рассмотрим представление деревьев с произвольным количеством дочерних элементов.

Каждый узел дерева представляет собой отдельный объект. Как и при изучении связанных списков, предполагается, что в каждом узле содержится атрибут key. Остальные интересующие нас атрибуты представляют собой указатели на другие узлы, и их вид зависит от типа дерева.

Бинарные деревья

Как показано на рис. 10.9, для хранения указателей на родительский, дочерний левый и дочерний правый узлы бинарного дерева T используются атрибуты p, $left$ и $right$. Если $x.p = \text{NIL}$, то x — корень дерева. Если у узла x нет левого дочернего узла, $x.left = \text{NIL}$; аналогичное утверждение справедливо в случае отсутствия правого дочернего узла. Атрибут $T.root$ указывает на корневой узел дерева T. Если $T.root = \text{NIL}$, то дерево T пустое.

Корневые деревья с произвольным ветвлением

Схему представления бинарных деревьев можно обобщить для деревьев любого класса, в которых количество дочерних узлов не превышает некоторой константы k. При этом атрибуты $left$ и $right$ заменяются атрибутами $child_1, child_2, \ldots, child_k$. Если количество дочерних элементов узла не ограничено, то эта схема не работает, поскольку заранее не известно, место для какого количества атрибутов (массивов при использовании представления с помощью нескольких массивов) нужно выделить. Кроме того, даже если количество дочерних элементов k ограничено большой константой, но на самом деле у многих узлов потомков намного меньше, то значительный объем памяти расходуется напрасно.

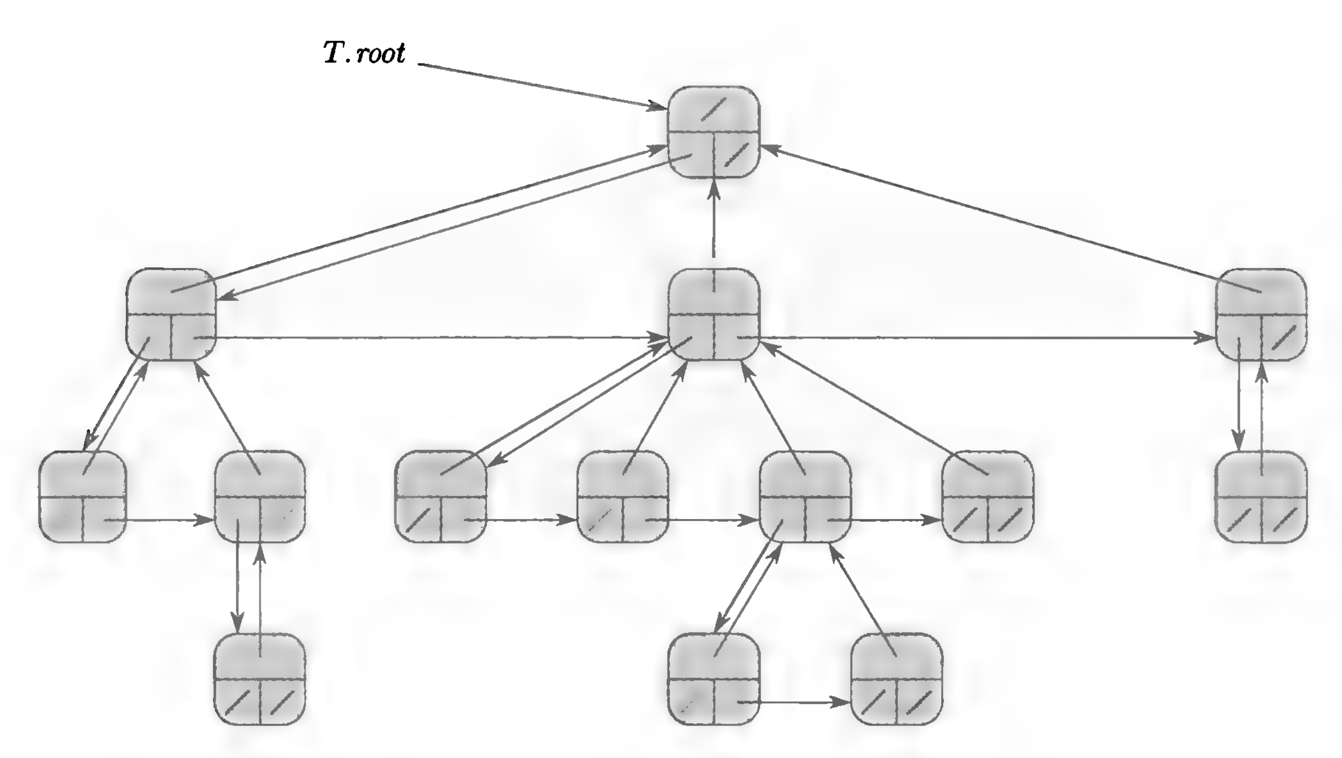


Рис. 10.10. Представление дерева T с левым дочерним и правым сестринским узлами. Каждый узел x имеет атрибуты $x.p$ (вверх), $x.\mathit{left\text{-}child}$ (влево вниз) и $x.\mathit{right\text{-}sibling}$ (вправо). Атрибуты key не показаны.

К счастью, существует остроумная схема представления деревьев с произвольным количеством дочерних узлов. Преимущество этой схемы в том, что для любого корневого дерева с n дочерними узлами требуется объем памяти $O(n)$. На рис. 10.10 проиллюстрировано представление ***с левым дочерним и правым сестринским узлами*** (left-child, right-sibling representation). Как и ранее, в каждом узле этого представления содержится указатель p на родительский узел, а атрибут $T.\mathit{root}$ указывает на корень дерева T. Однако вместо указателей на каждый дочерний узел каждый узел x содержит всего два указателя:

1. $x.\mathit{left\text{-}child}$ указывает на крайний слева дочерний узел узла x;
2. $x.\mathit{right\text{-}sibling}$ указывает на узел, расположенный на одном уровне с узлом x, справа от него и непосредственно рядом с ним.

Если узел x не имеет потомков, то $x.\mathit{left\text{-}child} = \text{NIL}$, а если узел x является крайним справа дочерним узлом своего родителя, то $x.\mathit{right\text{-}sibling} = \text{NIL}$.

Другие представления деревьев

Иногда встречаются и другие способы представления корневых деревьев. Например, в главе 6 описано представление пирамиды, основанной на полном бинарном дереве, с помощью одного массива и индекса последнего узла пирамиды. Деревья, о которых идет речь в главе 21, обходятся только по направлению к корню, поэтому в них содержатся лишь указатели на родительские элементы; указатели на дочерние элементы отсутствуют. Здесь возможны самые различные схемы. Наилучший выбор схемы зависит от конкретного приложения.

Упражнения

10.4.1

Начертите бинарное дерево, корень которого имеет индекс 6, и которое представлено приведенными ниже атрибутами.

Индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4.2

Разработайте рекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева.

10.4.3

Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. В качестве вспомогательной структуры данных используйте стек.

10.4.4

Разработайте процедуру, которая за время $O(n)$ выводит ключи всех n узлов произвольного корневого дерева. Дерево реализовано в представлении с левым дочерним и правым сестринским элементами.

10.4.5 ⋆

Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. Объем дополнительной памяти (кроме той, которая отводится под само дерево) не должен превышать некоторую константу. Кроме того, в процессе выполнения процедуры дерево (даже временно) не должно модифицироваться.

10.4.6 ⋆

В представлении произвольного корневого дерева с левым дочерним и правым сестринским узлами в каждом узле есть по три указателя: *left-child*, *right-sibling* и *parent*. Если задан какой-то узел дерева, то определить его родительский узел и получить к нему доступ можно в течение фиксированного времени. Определить же все дочерние узлы и получить к ним доступ можно в течение времени, линейно зависящего от количества дочерних узлов. Разработайте способ хранения дерева с произвольным ветвлением, в каждом узле которого используются только два (а не три) указателя и одно логическое значение, при котором ро-

дительский узел или все дочерние узлы идентифицируются в течение времени, линейно зависящего от количества дочерних узлов.

Задачи

10.1. Сравнение списков

Определите асимптотическое время выполнения перечисленных в приведенной ниже таблице операций над элементами динамических множеств в наихудшем случае, если эти операции выполняются со списками перечисленных ниже типов.

	Неотсортированный, однократно связанный список	Отсортированный, однократно связанный список	Неотсортированный, дважды связанный список	Отсортированный, дважды связанный список
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10.2. Реализация объединяемых пирамид с помощью связанных списков

В ***объединяемой пирамиде*** (mergable heap) поддерживаются следующие операции: MAKE-HEAP (создание пустой пирамиды), INSERT, MINIMUM, EXTRACT-MINIMUM и UNION[1]. Покажите, как в каждом из перечисленных ниже случаев реализовать объединяемые пирамиды с помощью связанных списков. Постарайтесь, чтобы каждая операция выполнялась с максимальной эффективностью. Проанализируйте время работы каждой операции по отношению к размеру обрабатываемого динамического множества.

а. Списки отсортированы.

б. Списки не отсортированы.

---

[1]Поскольку в пирамиде поддерживаются операции MINIMUM и EXTRACT-MINIMUM, такую пирамиду можно назвать ***объединяемой неубывающей пирамидой*** (mergable min-heap). Аналогично, если бы в пирамиде поддерживались операции MAXIMUM и EXTRACT-MAXIMUM, ее можно было бы назвать ***объединяемой невозрастающей пирамидой*** (mergable max-heap).

в. Списки не отсортированы, а объединяемые динамические множества не пересекаются.

10.3. Поиск в отсортированном компактном списке

В упр. 10.3.4 предлагается разработать компактную поддержку n-элементного списка в первых n позициях массива. Предполагается, что все ключи различны и что компактный список отсортирован, т.е. для всех $i = 1, 2, \ldots, n$, таких, что $next[i] \neq$ NIL, выполняется соотношение $key[i] < key[next[i]]$. Предполагается также, что имеется переменная L, в которой хранится индекс первого элемента списка. Покажите, что при данных предположениях математическое ожидание времени поиска с помощью приведенного ниже рандомизированного алгоритма равно $O(\sqrt{n})$.

```
COMPACT-LIST-SEARCH(L, n, k)
 1  i = L
 2  while i ≠ NIL и key[i] < k
 3      j = RANDOM(1, n)
 4      if key[i] < key[j] и key[j] ≤ k
 5          i = j
 6          if key[i] == k
 7              return i
 8      i = next[i]
 9  if i == NIL или key[i] > k
10      return NIL
11  else return i
```

Если в представленной выше процедуре опустить строки 3–7, получится обычный алгоритм, предназначенный для поиска в отсортированном связанном списке, в котором индекс i пробегает по очереди по всем элементам в списке. Поиск прекращается в тот момент, когда происходит "обрыв" индекса i в конце списка или когда $key[i] \geq k$. В последнем случае, если выполняется соотношение $key[i] = k$, понятно, что ключ k найден. Если же $key[i] > k$, то ключ k в списке отсутствует, и поэтому следует прекратить поиск.

В строках 3–7 предпринимается попытка перейти к случайно выбранной ячейке j. Такой переход дает преимущества, если величина $key[j]$ больше величины $key[i]$, но не превышает значения k. В этом случае индекс j соответствует элементу списка, к которому рано или поздно был бы осуществлен доступ при обычном поиске. Поскольку список компактен, любой индекс j в интервале от 1 до n отвечает некоторому объекту списка и не может быть пустым местом из списка свободных позиций.

Вместо производительности процедуры COMPACT-LIST-SEARCH мы проанализируем связанный с ним алгоритм COMPACT-LIST-SEARCH′, в котором содержатся два отдельных цикла. В этом алгоритме используется дополнительный параметр t, определяющий верхнюю границу количества итераций в первом цикле.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
1  i = L
2  for q = 1 to t
3      j = RANDOM(1, n)
4      if key[i] < key[j] и key[j] ≤ k
5          i = j
6          if key[i] == k
7              return i
8  while i ≠ NIL и key[i] < k
9      i = next[i]
10 if i == NIL или key[i] > k
11     return NIL
12 else return i
```

Для простоты сравнения алгоритмов COMPACT-LIST-SEARCH(L, n, k) и COMPACT-LIST-SEARCH'(L, n, k, t) будем считать, что последовательность целых чисел, которая возвращается вызовами RANDOM$(1, n)$, одна и та же для обоих алгоритмов.

а. Предположим, что в ходе работы цикла **while** в строках 2–8 процедуры COMPACT-LIST-SEARCH(L, n, k) выполняется t итераций. Докажите, что процедура COMPACT-LIST-SEARCH'(L, n, k, t) даст тот же ответ и что общее количество итераций в циклах **for** и **while** процедуры COMPACT-LIST-SEARCH' не меньше t.

Пусть X_t — случайная величина, описывающая расстояние в связанном списке (т.е. длину цепочки из указателей *next*) от элемента с индексом i до искомого ключа k после t итераций цикла **for** в строках 2–7 в вызове процедуры COMPACT-LIST-SEARCH'(L, n, k, t) .

б. Докажите, что математическое ожидание времени работы процедуры COMPACT-LIST-SEARCH'(L, n, k, t) равно $O(t + \mathrm{E}[X_t])$.

в. Покажите, что $\mathrm{E}[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Указание:* воспользуйтесь уравнением (В.25).)

г. Покажите, что $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.

д. Докажите, что $\mathrm{E}[X_t] \leq n/(t+1)$.

е. Покажите, что математическое ожидание времени работы процедуры COMPACT-LIST-SEARCH'(L, n, k, t) равно $O(t + n/t)$.

ж. Сделайте вывод о том, что математическое ожидание времени работы процедуры COMPACT-LIST-SEARCH равно $O(\sqrt{n})$.

з. Объясните, зачем в ходе анализа процедуры COMPACT-LIST-SEARCH понадобилось предположение о том, что все ключи различны. Покажите, что случайные переходы не обязательно приведут к сокращению асимптотического времени работы, если в списке содержатся ключи с одинаковыми значениями.

Заключительные замечания

Прекрасными справочными пособиями по элементарным структурам данных являются книги Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [6] и Кнута (Knuth) [208][2]. Описание базовых структур данных и их реализации в конкретных языках программирования можно также найти во многих других учебниках. В качестве примера можно привести книги Гудрича (Goodrich) и Тамазии (Tamassia) [146], Мейна (Main) [240], Шаффера (Shaffer) [309] и Вейса (Weiss) [350–352]. В книге Гонне (Gonnet) [144] приведены экспериментальные данные, описывающие производительность операций над многими структурами данных.

Начало использования стеков и очередей в информатике в качестве структур данных по сей день остается недостаточно ясным. Вопрос усложняется тем, что соответствующие понятия существовали в математике и применялись на практике при выполнении деловых расчетов на бумаге еще до появления первых цифровых вычислительных машин. В своей книге [208] Кнут приводит относящиеся к 1947 году цитаты А.М. Тьюринга (A.M. Turing), в которых идет речь о разработке стеков для компоновки подпрограмм.

По-видимому, структуры данных, основанные на применении указателей, также являются "народным" изобретением. Согласно Кнуту указатели, скорее всего, использовались еще на первых компьютерах с памятью барабанного типа. В языке программирования А-1, разработанном Г.М. Хоппером (G.M. Hopper) в 1951 году, алгебраические формулы представляются в виде бинарных деревьев. Кнут считает, что указатели получили признание и широкое распространение благодаря языку программирования IPL-II, разработанному в 1956 году А. Ньювеллом (A. Newell), Д.К. Шоу (J.C. Shaw) и Г.А. Саймоном (H.A. Simon). В язык IPL-III, разработанный в 1957 году теми же авторами, операции со стеком включены явным образом.

[2]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 11. Хеширование и хеш-таблицы

Для многих приложений достаточно динамических множеств, поддерживающих только стандартные словарные операции INSERT, SEARCH и DELETE. Например, компилятор, транслирующий язык программирования, поддерживает таблицу символов, в которой ключами элементов являются произвольные символьные строки, соответствующие идентификаторам в языке. Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, сколько на поиск в связанном списке, а именно — $\Theta(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях среднее время поиска элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица обобщает обычный массив. Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Более подробно прямая индексация рассматривается в разделе 11.1; она применима, если мы в состоянии выделить массив такого размера, какого достаточно для того, чтобы для каждого возможного значения ключа имелась своя ячейка.

Если количество реально хранящихся в массиве ключей мало́ по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив, размер которого пропорционален количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива индекс *вычисляется* по значению ключа. В разделе 11.2 представлены основные идеи хеширования, в первую очередь направленные на разрешение коллизий (когда несколько ключей отображается в один и тот же индекс массива) с помощью цепочек. В разделе 11.3 описывается, каким образом на основе значений ключей могут быть вычислены индексы массива. Здесь будет рассмотрено и проанализировано несколько вариантов хеш-функций. В разделе 11.4 вы познакомитесь с методом открытой адресации, представляющим собой еще один способ разрешения коллизий. Главный вывод, который следует из всего изложенного материала, — хеширование представляет собой исключительно эффективную и практичную технологию: в среднем все базовые словарные операции выполняются за время $O(1)$. В разделе 11.5 будет дано пояснение, каким образом "идеальное хеширование" может поддерживать *наихудшее* время поиска $O(1)$ в случае ис-

пользования статического множества хранящихся ключей (т.е. когда множество ключей, будучи сохраненным, более не изменяется).

11.1. Таблицы с прямой адресацией

Прямая адресация представляет собой простую технологию, которая хорошо работает для небольших совокупностей ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из совокупности $U = \{0, 1, \dots, m-1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или ***таблицу с прямой адресацией***, который обозначим как $T[0\,..\,m-1]$, каждая позиция (position), или ячейка, слот (slot) которого соответствует ключу из совокупности ключей U. На рис. 11.1 проиллюстрирован данный подход. Ячейка k указывает на элемент множества с ключом k. Если множество не содержит элемента с ключом k, то $T[k] = \text{NIL}$.

Реализация словарных операций тривиальна.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Каждая из этих операций выполняется за время $O(1)$.

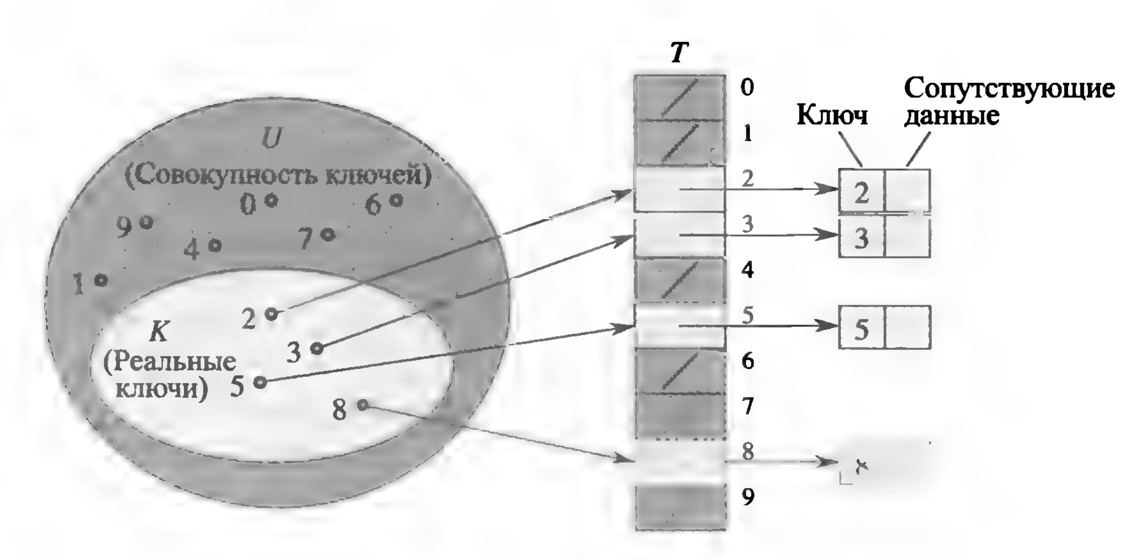


Рис. 11.1. Реализация динамического множества с использованием таблицы с прямой адресацией T. Каждый ключ в совокупности $U = \{0, 1, \dots, 9\}$ соответствует индексу в таблице. Множество $K = \{2, 3, 5, 8\}$ реальных ключей определяет ячейки в таблице, которые содержат указатели на элементы. Прочие (закрашенные темным цветом) ячейки содержат значение NIL.

В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. Иначе говоря, вместо хранения ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице — указателей на эти объекты, эти объекты можно хранить непосредственно в ячейках таблицы (что тем самым приводит к экономии используемой памяти). При этом для указания пустой ячейки можно воспользоваться специальным значением ключа. Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если мы знаем индекс объекта в таблице, значит, мы знаем и его ключ. Однако если ключ не хранится в ячейке таблицы, то нам нужен какой-то иной механизм для того, чтобы помечать пустые ячейки.

Упражнения

11.1.1

Предположим, что динамическое множество S представлено таблицей с прямой адресацией T длиной m. Опишите процедуру, которая находит максимальный элемент S. Чему равно время работы этой процедуры в наихудшем случае?

11.1.2

Битовый вектор представляет собой массив битов (нулей и единиц). Битовый вектор длиной m занимает существенно меньше места, чем массив из m указателей. Каким образом можно использовать битовый вектор для представления динамического множества различных элементов без сопутствующих данных? Словарные операции должны выполняться за время $O(1)$.

11.1.3

Предложите способ реализации таблицы с прямой адресацией, в которой ключи хранящихся элементов могут совпадать, а сами элементы — иметь сопутствующие данные. Все словарные операции (INSERT, DELETE и SEARCH) должны выполняться за время $O(1)$. (Не забудьте, что аргументом процедуры DELETE является указатель на удаляемый объект, а не ключ.)

11.1.4 ★

Предположим, что мы хотим реализовать словарь с использованием прямой адресации *очень большого* массива. Первоначально в массиве может содержаться "мусор", но инициализация всего массива нерациональна в силу его размера. Разработайте схему реализации словаря с прямой адресацией при описанных условиях. Каждый хранимый объект должен использовать $O(1)$ памяти; операции SEARCH, INSERT и DELETE должны выполняться за время $O(1)$; инициализация структуры данных также должна выполняться за время $O(1)$. (*Указание:* для определения, является ли данная запись в большом массиве корректной, воспользуйтесь дополнительным массивом, работающим в качестве стека, размер которого равен количеству ключей, сохраненных в словаре.)

11.2. Хеш-таблицы

Недостаток прямой адресации очевиден: если совокупность ключей U велика, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно — в зависимости от количества доступной памяти и размера совокупности ключей. Кроме того, множество K *реально сохраненных* ключей может быть мало́ по сравнению с совокупностью ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Когда множество K хранящихся в словаре ключей гораздо меньше совокупности возможных ключей U, для хеш-таблицы требуется существенно меньше места, чем для таблицы с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Нужно только заметить, что это граница времени поиска *в среднем случае*, в то время как в случае таблицы с прямой адресацией эта граница справедлива для *наихудшего случая*.

В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем ***хеш-функцию*** h для вычисления ячейки для данного ключа k. Функция h отображает совокупность ключей U на ячейки ***хеш-таблицы*** $T[0..m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\} ,$$

где размер m хеш-таблицы обычно гораздо меньше значения $|U|$. Мы говорим, что элемент с ключом k ***хешируется*** в ячейку $h(k)$; величина $h(k)$ называется ***хеш-значением*** ключа k. На рис. 11.2 представлена основная идея хеширования. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо размера $|U|$ значений мы можем обойтись массивом всего лишь размером m.

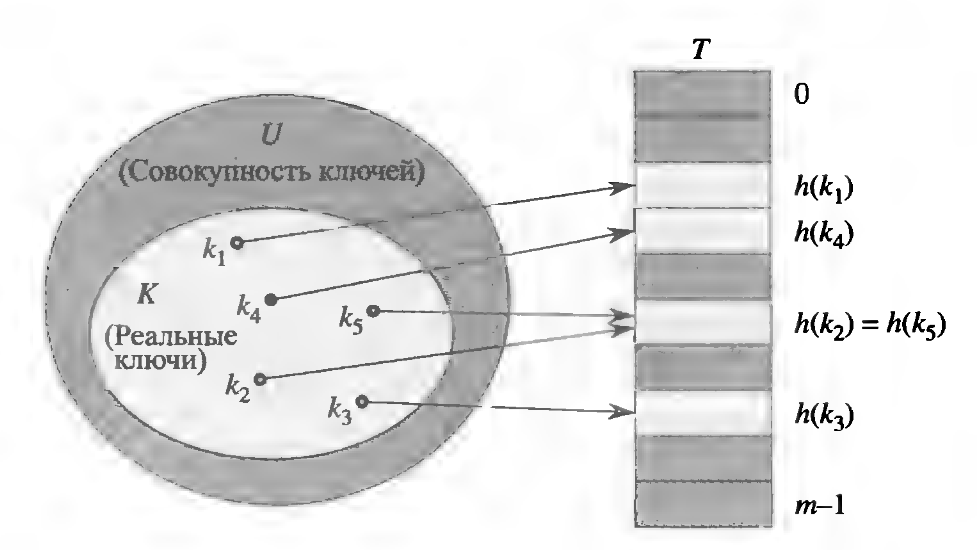


Рис. 11.2. Применение хеш-функции h для отображения ключей в ячейки хеш-таблицы. Ключи k_2 и k_5 отображаются в одну ячейку, вызывая коллизию.

Однако здесь есть одна проблема: два ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется ***коллизией***. К счастью, имеются эффективные технологии разрешения конфликтов, вызываемых коллизиями.

Конечно, идеальным решением было бы полное устранение коллизий. Мы можем попытаться добиться этого путем выбора подходящей хеш-функции h. Одна из идей заключается в том, чтобы сделать функцию h "случайной", что позволило бы избежать коллизий или хотя бы минимизировать их количество (этот характер функции хеширования отображается в самом глаголе "to hash", который означает "мелко порубить, перемешать"). Само собой разумеется, функция h должна быть детерминистической и для одного и того же значения k всегда давать одно и то же хеш-значение $h(k)$. Однако поскольку $|U| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью избежать коллизий невозможно в принципе, и хорошая хеш-функция в состоянии только минимизировать их количество. Таким образом, нам все равно нужен метод разрешения возникающих коллизий.

В оставшейся части данного раздела мы рассмотрим простейший метод разрешения коллизий — метод цепочек. В разделе 11.4 вы познакомитесь с еще одним методом разрешения коллизий, который называется методом открытой адресации.

Разрешение коллизий с помощью цепочек

При разрешении коллизий ***с помощью цепочек*** мы помещаем все элементы, хешированные в одну и ту же ячейку, в связанный список, как показано на рис. 11.3. Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение NIL.

Словарные операции в хеш-таблице с использованием цепочек для разрешения коллизий реализуются очень просто.

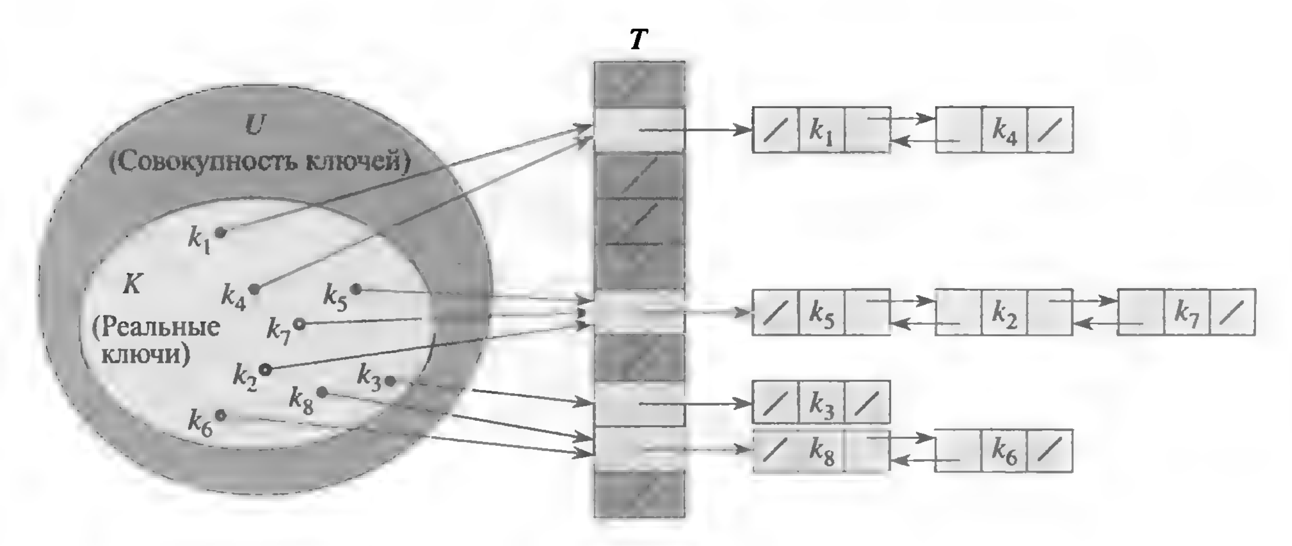


Рис. 11.3. Разрешение коллизий с помощью цепочек. Каждая ячейка хеш-таблицы $T[j]$ содержит связанный список всех ключей с хеш-значением j. Например, $h(k_1) = h(k_4)$ и $h(k_5) = h(k_7) = h(k_2)$. Связанный список может быть одинарно или дважды связанным; мы показываем его как дважды связанный, поскольку удаление в этом случае выполняется гораздо быстрее.

CHAINED-HASH-INSERT(T, x)
1 Вставка x в заголовок списка $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
1 Поиск элемента с ключом k в списке $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
1 Удаление x из списка $T[h(x.key)]$

Время, необходимое для вставки в наихудшем случае, равно $O(1)$. Процедура вставки выполняется очень быстро, в частности, потому, что предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено дополнительной ценой выполнения поиска элемента с ключом $x.key$ перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка; мы проанализируем эту операцию немного позже. Удаление элемента может быть выполнено за время $O(1)$ при использовании дважды связанных списков, как на рис. 11.3. (Обратите внимание на то, что процедура CHAINED-HASH-DELETE принимает в качестве аргумента элемент x, а не его ключ, поэтому нет необходимости в предварительном поиске x. Если хеш-таблица поддерживает удаление, ее списки должны быть двусвязными для ускорения процесса удаления. Если список односвязный, то передача в качестве аргумента x не дает нам особого выигрыша, поскольку для корректного обновления атрибута *next* предшественника x нам все равно нужно выполнить поиск x в списке $T[h(x.key)]$. В таком случае, как нетрудно понять, удаление и поиск имеют, по сути, одно и то же асимптотическое время работы.)

Анализ хеширования с цепочками

Насколько высока производительность хеширования с цепочками? В частности, сколько времени требуется для поиска элемента с заданным ключом?

Пусть у нас есть хеш-таблица T с m ячейками, в которых хранятся n элементов. Определим ***коэффициент заполнения*** α таблицы T как n/m, т.е. как среднее количество элементов, хранящихся в одной цепочке. Наш анализ будет опираться на значение величины α, которая может быть меньше, равна или больше единицы.

В наихудшем случае хеширование с цепочками ведет себя крайне неприятно: все n ключей хешированы в одну и ту же ячейку, создав список длиной n. Таким образом, время поиска в наихудшем случае равно $\Theta(n)$ плюс время вычисления хеш-функции, что ничуть не лучше, чем в случае использования связанного списка для хранения всех n элементов. Понятно, что использование хеш-таблиц в наихудшем случае совершенно бессмысленно. (Идеальное хеширование, применимое в случае статического множества ключей и рассмотренное в разделе 11.5, обеспечивает высокую производительность даже в наихудшем случае.)

Производительность хеширования в среднем случае зависит от того, насколько хорошо хеш-функция h распределяет множество сохраняемых ключей по m ячейкам в среднем. Мы рассмотрим этот вопрос подробнее в разделе 11.3, а пока

будем полагать, что все элементы хешируются по ячейкам равномерно и независимо, и назовем данное предположение ***простым равномерным хешированием*** (simple uniform hashing).

Обозначим длины списков $T[j]$ для $j = 0, 1, \ldots, m-1$ как n_j, так что

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

а ожидаемое значение n_j равно $\mathrm{E}[n_j] = \alpha = n/m$.

Мы полагаем, что для вычисления хеш-значения $h(k)$ достаточно времени $O(1)$, так что время, необходимое для поиска элемента с ключом k, линейно зависит от длины $n_{h(k)}$ списка $T[h(k)]$. Не учитывая время $O(1)$, требующееся для вычисления хеш-функции и доступа к ячейке $h(k)$, рассмотрим математическое ожидание количества элементов, которое должно быть проверено алгоритмом поиска (т.е. количество элементов в списке $T[h(k)]$, которые проверяются на равенство их ключей величине k). Мы должны рассмотреть два случая: во-первых, когда поиск неудачен и в таблице нет элементов с ключом k и, во-вторых, когда поиск заканчивается успешно и в таблице определяется элемент с ключом k.

Теорема 11.1

В хеш-таблице с разрешением коллизий методом цепочек время неудачного поиска в среднем случае в предположении простого равномерного хеширования составляет $\Theta(1+\alpha)$.

Доказательство. В предположении простого равномерного хеширования любой ключ k, который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка $T[h(k)]$, ожидаемая длина которого — $\mathrm{E}\left[n_{h(k)}\right] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α, а общее время, необходимое для поиска, включая время вычисления хеш-функции $h(k)$, равно $\Theta(1+\alpha)$. ■

Успешный поиск несколько отличается от неудачного, поскольку вероятность поиска в списке различна для разных списков и пропорциональна количеству содержащихся в нем элементов. Тем не менее и в этом случае математическое ожидание времени поиска остается равным $\Theta(1+\alpha)$.

Теорема 11.2

В хеш-таблице с разрешением коллизий методом цепочек время успешного поиска в среднем случае в предположении простого равномерного хеширования в среднем равно $\Theta(1+\alpha)$.

Доказательство. Мы полагаем, что искомый элемент с равной вероятностью может быть любым элементом, хранящимся в таблице. Количество элементов, проверяемых в процессе успешного поиска элемента x, на 1 больше, чем количество элементов, находящихся в списке перед x. Элементы, находящиеся в списке

до x, были вставлены в список после того, как элемент x был сохранен в таблице, так как новые элементы помещаются в начало списка. Для того чтобы найти математическое ожидание количества проверяемых элементов, мы возьмем среднее по всем n элементам x в таблице значение, которое равно 1 плюс математическое ожидание количества элементов, добавленных в список x после самого́ искомого элемента. Пусть x_i обозначает i-й элемент, вставленный в таблицу ($i = 1, 2, \ldots, n$), и пусть $k_i = x_i.key$. Определим для ключей k_i и k_j индикаторную случайную величину $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$. В предположении простого равномерного хеширования $\Pr\{h(k_i) = h(k_j)\} = 1/m$ и, в соответствии с леммой 5.1, $\mathrm{E}[X_{ij}] = 1/m$. Таким образом, математическое ожидание количества проверяемых элементов в случае успешного поиска равно

$$
\begin{aligned}
\mathrm{E}&\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(из линейности математического ожидания)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(согласно (A.1))} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Таким образом, полное время, необходимое для проведения успешного поиска (включая время вычисления хеш-функции), составляет $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ∎

О чем говорит проведенный анализ? Если количество ячеек в хеш-таблице как минимум пропорционально количеству элементов, хранящихся в ней, то $n = O(m)$ и, следовательно, $\alpha = n/m = O(m)/m = O(1)$. Таким образом, поиск элемента в хеш-таблице в среднем требует постоянного времени. Поскольку в худшем случае вставка элемента в хеш-таблицу занимает $O(1)$ времени (как и удаление элемента при использовании дважды связанных списков), можно сделать вывод, что все словарные операции в хеш-таблице в среднем выполняются за время $O(1)$.

Упражнения

11.2.1

Предположим, что мы используем хеш-функцию h для хеширования n различных ключей в массив T, длина которого равна m. Чему равно математическое ожидание количества коллизий в предположении простого равномерного хеширования? Говоря точнее, какова ожидаемая мощность множества $\{\{k, l\} : k \neq l$ и $h(k) = h(l)\}$?

11.2.2

Продемонстрируйте происходящее при вставке в хеш-таблицу с разрешением коллизий методом цепочек ключей $5, 28, 19, 15, 20, 33, 12, 17, 10$. Таблица имеет 9 ячеек, а хеш-функция имеет вид $h(k) = k \bmod 9$.

11.2.3

Профессор выдвинул гипотезу о том, что можно существенно повысить эффективность хеширования с разрешением коллизий методом цепочек, если поддерживать списки в упорядоченном состоянии. Каким образом такое изменение алгоритма повлияет на время выполнения успешного поиска, неудачного поиска, вставки и удаления?

11.2.4

Предложите способ выделения и освобождения элементов внутри самóй хеш-таблицы, при котором неиспользуемые ячейки связываются в один список свободных мест. Считается, что в каждой ячейке может храниться флаг и либо один элемент и указатель, либо два указателя. Все словарные операции и операции над списком свободных мест должны выполняться за ожидаемое время $O(1)$. Должен ли список свободных мест быть двусвязным или можно обойтись односвязным списком?

11.2.5

Предположим, что необходимо хранить множество из n ключей в хеш-таблице размером m. Покажите, что если ключи выбираются из совокупности U с $|U| > nm$, то U имеет подмножество размером n, состоящее из ключей, хешируемых в одну и ту же ячейку, так что время поиска при хешировании с цепочками в наихудшем случае составляет $\Theta(n)$.

11.2.6

Предположим, что необходимо хранить множество из n ключей в хеш-таблице размером m, с разрешением коллизий методом цепочек, и что известна длина каждой цепочки, включая длину L самой длинной из них. Опишите процедуру, которая случайным образом выбирает ключ среди всех ключей таблицы за ожидаемое время $O(L \cdot (1 + 1/\alpha))$ (все вероятности выбора каждого из ключей одинаковы).

11.3. Хеш-функции

В этом разделе мы рассмотрим некоторые вопросы, связанные с разработкой качественных хеш-функций, и познакомимся с тремя схемами их построения. Две из них, хеширование делением и хеширование умножением, эвристичны по своей природе, в то время как третья схема — универсальное хеширование — использует рандомизацию для обеспечения доказуемо высокой производительности.

Чем определяется качество хеш-функции

Качественная хеш-функция удовлетворяет (приближенно) предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из m ячеек независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно. Более того, вставляемые ключи могут не быть независимыми.

Иногда распределение вероятностей оказывается известным. Например, если известно, что ключи представляют собой случайные действительные числа, равномерно распределенные в диапазоне $0 \le k < 1$, то хеш-функция

$$h(k) = \lfloor km \rfloor$$

удовлетворяет условию простого равномерного хеширования.

На практике при построении качественных хеш-функций зачастую используются различные эвристические методики. В процессе построения большую помощь оказывает информация о распределении ключей. Рассмотрим, например, таблицу символов компилятора, в которой ключами служат символьные строки, представляющие идентификаторы в программе. Зачастую в одной программе встречаются похожие идентификаторы, например `pt` и `pts`. Хорошая хеш-функция должна минимизировать шансы попадания этих идентификаторов в одну ячейку хеш-таблицы.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак не коррелировала с закономерностями, которым могут подчиняться существующие данные. Например, метод деления, который рассматривается в разделе 11.3.1, вычисляет хеш-значение как остаток от деления ключа на некоторое простое число. Если это простое число никак не связано с распределением исходных данных, метод часто дает хорошие результаты.

В заключение заметим, что некоторые приложения хеш-функций могут накладывать более строгие требования по сравнению с требованиями простого равномерного хеширования. Например, мы можем потребовать, чтобы "близкие" в некотором смысле ключи давали далекие хеш-значения (это свойство особенно желательно при использовании линейного исследования, описанного в разделе 11.4). Универсальное хеширование, описанное в разделе 11.3.3, часто приводит к желаемым результатам.

Интерпретация ключей как целых неотрицательных чисел

Для большинства хеш-функций совокупность ключей представляется множеством целых неотрицательных чисел $\mathbb{N} = \{0, 1, 2, \ldots\}$. Если же ключи не являются целыми неотрицательными числами, то можно найти способ их интерпретации как таковых. Например, строка символов может рассматриваться как целое число, записанное в соответствующей системе счисления. Так, идентификатор `pt` можно рассматривать как пару десятичных чисел $(112, 116)$, поскольку в ASCII-наборе символов `p` $= 112$ и `t` $= 116$. Рассматривая `pt` как число в системе счисления с основанием 128, мы находим, что оно соответствует значению $(112 \cdot 128) + 116 = 14452$. В конкретных приложениях обычно не представляет особого труда разработать метод для представления ключей в виде (возможно, бóльших) целых чисел. Далее при изложении материала мы будем считать, что все ключи представляют собой целые неотрицательные числа.

11.3.1. Метод деления

Построение хеш-функции ***методом деления*** состоит в отображении ключа k в одну из m ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид

$$h(k) = k \bmod m \ .$$

Например, если хеш-таблица имеет размер $m = 12$, а значение ключа $k = 100$, то $h(k) = 4$. Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления достаточно быстрое.

При использовании данного метода мы обычно стараемся избегать некоторых значений m. Например, m не должно быть степенью 2, поскольку если $m = 2^p$, то $h(k)$ представляет собой просто p младших битов числа k. Если только заранее не известно, что все наборы младших p битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа. В упр. 11.3.3 требуется показать неудачность выбора $m = 2^p - 1$, когда ключи представляют собой строки символов, интерпретируемые как числа в системе счисления с основанием 2^p, поскольку перестановка символов ключа не приводит к изменению его хеш-значения.

Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки. Предположим, например, что мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения порядка $n = 2000$ символьных строк, размер символов в которых равен 8 бит. Нас устраивает проверка в среднем трех элементов при неудачном поиске, так что мы выбираем размер таблицы равным $m = 701$. Число 701 выбрано как простое число, близкое к величине $2000/3$ и не являющееся степенью 2. Рассматривая каждый ключ k как целое число, мы получаем искомую хеш-функцию

$$h(k) = k \bmod 701 \ .$$

11.3.2. Метод умножения

Построение хеш-функции ***методом умножения*** выполняется в два этапа. Сначала мы умножаем ключ k на константу $0 < A < 1$ и выделяем дробную часть полученного произведения. Затем мы умножаем полученное значение на m и применяем к нему функцию "пол". Короче говоря, хеш-функция имеет вид

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor \ ,$$

где выражение "$kA \bmod 1$" означает получение дробной части произведения kA, т.е. величину $kA - \lfloor kA \rfloor$.

Преимущество метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2 ($m = 2^p$ для некоторого натурального p), поскольку такая функция легко реализуема на большинстве компьютеров. Пусть у нас имеется компьютер с размером слова w бит и k помещается в одно слово. Ограничим возможные значения константы A дробями вида $s/2^w$, где s — целое число из диапазона $0 < s < 2^w$. Тогда мы сначала умножаем k на w-битовое целое число $s = A \cdot 2^w$. Результат представляет собой $2w$-битовое число $r_1 2^w + r_0$, где r_1 — старшее слово произведения, а r_0 — младшее. Старшие p бит числа r_0 представляют собой искомое p-битовое хеш-значение (рис. 11.4).

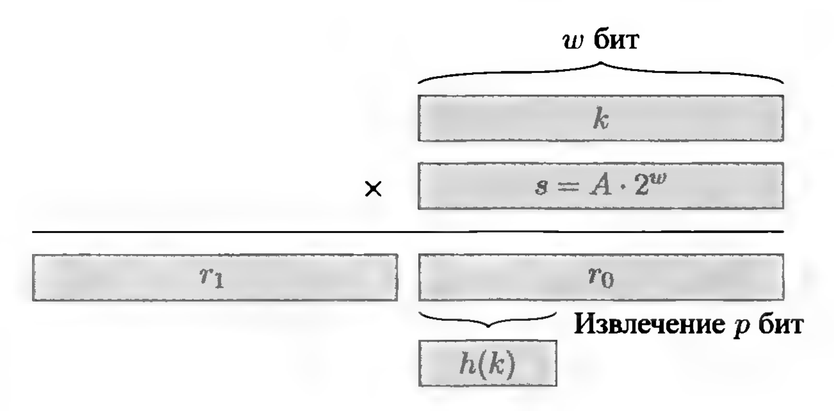


Рис. 11.4. Хеширование методом умножения. w-битовое представление ключа k умножается на w-битовое значение $s = A \cdot 2^w$. p старших битов младшей w-битовой половины произведения образует искомое хеш-значение $h(k)$

Хотя описанный метод работает с любыми значениями константы A, одни значения дают лучшие результаты по сравнению с другими. Оптимальный выбор зависит от характеристик хешируемых данных. В [210][1] Кнут (Knuth) предложил использовать дающее неплохие результаты значение

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots \qquad (11.2)$$

В качестве примера предположим, что $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ и $w = 32$. Принимая предложения Кнута, выберем A в виде дроби $s/2^{32}$, бли-

[1]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

жайшей к значению $(\sqrt{5}-1)/2$, так что $A = 2654435769/2^{32}$. Тогда $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, и, таким образом, $r_1 = 76300$ и $r_0 = 17612864$. Старшие 14 бит числа r_0 дают хеш-значение $h(k) = 67$.

★ 11.3.3 Универсальное хеширование

Если недоброжелатель будет умышленно выбирать ключи для хеширования с использованием конкретной хеш-функции, то он сможет подобрать n значений, которые будут хешироваться в одну и ту же ячейку таблицы, приводя к среднему времени выборки $\Theta(n)$. Таким образом, любая фиксированная хеш-функция становится уязвимой, и единственный эффективный выход из ситуации — *случайный* выбор хеш-функции, *не зависящий* от того, с какими именно ключами ей предстоит работать. Такой подход, который называется ***универсальным хешированием***, гарантирует хорошую производительность в среднем, независимо от того, какие данные будут выбраны упомянутым недоброжелателем.

Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Как и в случае быстрой сортировки, рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз по-разному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных. Возвращаясь к примеру с таблицей символов компилятора, мы обнаружим, что никакой выбор программистом имен идентификаторов не может привести к постоянно низкой производительности хеширования. Такое снижение возможно только тогда, когда компилятором выбрана случайная хеш-функция, которая приводит к плохому хешированию конкретных входных данных; однако вероятность такой ситуации очень мала и одинакова для любого множества идентификаторов одного и то же размера.

Пусть $\mathcal{H}$ — конечное множество хеш-функций, которые отображают данную совокупность ключей U в диапазон $\{0, 1, \ldots, m-1\}$. Такое множество называется ***универсальным***, если для каждой пары различных ключей $k, l \in U$ количество хеш-функций $h \in \mathcal{H}$, для которых $h(k) = h(l)$, не превышает $|\mathcal{H}|/m$. Другими словами, при случайном выборе хеш-функции из $\mathcal{H}$ вероятность коллизии между различными ключами k и l не превышает вероятности совпадения двух случайным образом выбранных хеш-значений из множества $\{0, 1, \ldots, m-1\}$, которая равна $1/m$.

Следующая теорема показывает, что универсальный класс хеш-функций обеспечивает хорошую среднюю производительность. В приведенной теореме n_i, как уже упоминалось, обозначает длину списка $T[i]$.

Теорема 11.3

Пусть хеш-функция h, случайным образом выбранная из универсального множества хеш-функций, применяется для хеширования n ключей в таблицу T размером m с использованием для разрешения коллизий метода цепочек. Если ключ k отсутствует в таблице, то математическое ожидание $\mathrm{E}\left[n_{h(k)}\right]$ длины списка, в ко-

торый хешируется ключ k, не превышает коэффициента заполнения $\alpha = n/m$. Если ключ k находится в таблице, то математическое ожидание $\mathrm{E}\left[n_{h(k)}\right]$ длины списка, в котором находится ключ k, не превышает $1 + \alpha$.

Доказательство. Заметим, что математическое ожидание вычисляется на множестве выборов функций и не зависит от каких бы то ни было предположений о распределении ключей. Определим для каждой пары различных ключей k и l индикаторную случайную величину $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Поскольку по определению универсального множества пара ключей вызывает коллизию с вероятностью не выше $1/m$, получаем, что $\Pr\{h(k) = h(l)\} \le 1/m$. Следовательно, в соответствии с леммой 5.1 мы имеем $\mathrm{E}[X_{kl}] \le 1/m$.

Далее для каждого ключа k определим случайную величину Y_k, которая равна количеству ключей, отличающихся от k и хешируемых в ту же ячейку, что и ключ k, так что

$$Y_k = \sum_{\substack{l \in T \\ l \ne k}} X_{kl} .$$

Таким образом, мы имеем

$$\begin{aligned}
\mathrm{E}[Y_k] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \ne k}} X_{kl}\right] \\
&= \sum_{\substack{l \in T \\ l \ne k}} \mathrm{E}[X_{kl}] && \text{(из линейности математического ожидания)} \\
&\le \sum_{\substack{l \in T \\ l \ne k}} \frac{1}{m} .
\end{aligned}$$

Оставшаяся часть доказательства зависит от того, находится ли ключ k в таблице T.

- Если $k \notin T$, то $n_{h(k)} = Y_k$ и $|\{l : l \in T \text{ и } l \ne k\}| = n$. Таким образом, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}[Y_k] \le n/m = \alpha$.
- Если $k \in T$, то, поскольку ключ k находится в списке $T[h(k)]$ и количество Y_k не включает ключ k, мы имеем $n_{h(k)} = Y_k + 1$ и $|\{l : l \in T \text{ и } l \ne k\}| = n - 1$. Таким образом, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}[Y_k]+1 \le (n-1)/m+1 = 1+\alpha-1/m < 1+\alpha$. ∎

Следствие из данной теоремы гласит, что универсальное хеширование обеспечивает желаемый выигрыш: теперь невозможно выбрать последовательность операций, которые приведут к наихудшему времени работы. Путем рандомизации выбора хеш-функции в процессе работы программы гарантируется хорошее среднее время работы алгоритма для любых входных данных.

Следствие 11.4

Использование универсального хеширования и разрешения коллизий методом цепочек в изначально пустой хеш-таблице с m ячейками дает математическое ожидание времени выполнения любой последовательности из n вызовов INSERT, SEARCH и DELETE, в которой содержится $O(m)$ операций INSERT, равное $\Theta(n)$.

Доказательство. Поскольку количество вставок равно $O(m)$, мы имеем $n = O(m)$ и, соответственно, $\alpha = O(1)$. Время работы операций INSERT и DELETE — величина постоянная, а в соответствии с теоремой 11.3 математическое ожидание времени выполнения каждой операции поиска равно $O(1)$. Таким образом, используя свойство линейности математического ожидания, получаем, что ожидаемое время, необходимое для выполнения всей последовательности из n операций, равно $O(n)$. Поскольку каждая операция занимает время $\Omega(1)$, отсюда следует граница $\Theta(n)$. ■

Построение универсального класса хеш-функций

Построить такое множество довольно просто, что следует из теории чисел. Если вы с ней незнакомы, то можете сначала обратиться к главе 31.

Начнем с выбора простого числа p, достаточно большого, чтобы все возможные ключи находились в диапазоне от 0 до $p-1$ включительно. Пусть $\mathbb{Z}_p$ обозначает множество $\{0, 1, \ldots, p-1\}$, а $\mathbb{Z}_p^*$ — множество $\{1, 2, \ldots, p-1\}$. Поскольку p — простое число, мы можем решать уравнения по модулю p с помощью методов, описанных в главе 31. Из предположения о том, что пространство ключей больше, чем количество ячеек в хеш-таблице, следует, что $p > m$.

Теперь определим хеш-функцию h_{ab} для любых $a \in \mathbb{Z}_p^*$ и $b \in \mathbb{Z}_p$, используя линейное преобразование с последующим приведением по модулю p, а затем по модулю m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \ . \tag{11.3}$$

Например, при $p = 17$ и $m = 6$ мы имеем $h_{3,4}(8) = 5$. Семейство всех таких функций образует множество

$$\mathcal{H}_{pm} = \left\{h_{ab} : a \in \mathbb{Z}_p^* \text{ и } b \in \mathbb{Z}_p\right\} \ . \tag{11.4}$$

Каждая хеш-функция h_{ab} отображает $\mathbb{Z}_p$ в $\mathbb{Z}_m$. Этот класс хеш-функций обладает тем свойством, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Это свойство будет использовано нами в разделе 11.5. Поскольку число a можно выбрать $p-1$ способом, а число b — p способами, всего во множестве $\mathcal{H}_{pm}$ содержится $p(p-1)$ хеш-функций.

Теорема 11.5

Класс $\mathcal{H}_{pm}$ хеш-функций, определяемых уравнениями (11.3) и (11.4), является универсальным.

Доказательство. Рассмотрим два различных ключа k и l из $\mathbb{Z}_p$, так что $k \neq l$. Пусть для данной хеш-функции h_{ab}

$$r = (ak + b) \bmod p \ ,$$
$$s = (al + b) \bmod p \ .$$

Заметим сначала, что $r \neq s$. Почему? Обратите внимание, что

$$r - s \equiv a(k - l) \pmod p \ .$$

Отсюда следует, что $r \neq s$, поскольку p — простое число, и как a, так и $(k - l)$ ненулевые по модулю p, так что и их произведение должно быть ненулевым по модулю p согласно теореме 31.6. Следовательно, вычисление любой хеш-функции $h_{ab} \in \mathcal{H}_{pm}$ отображает различные входные значения k и l на различные значения r и s по модулю p; так что коллизии "по модулю p" отсутствуют. Более того, каждый из $p(p - 1)$ возможных выборов пар (a, b) с $a \neq 0$ дает *различные* пары (r, s) с $r \neq s$, поскольку по заданным r и s можно найти a и b:

$$a = \big((r - s)((k - l)^{-1} \bmod p)\big) \bmod p \ ,$$
$$b = (r - ak) \bmod p \ ,$$

где $((k - l)^{-1} \bmod p)$ обозначает единственное мультипликативное обратное по модулю p значения $k - l$. Поскольку имеется только $p(p - 1)$ возможных пар (r, s), таких что $r \neq s$, то имеется взаимно однозначное соответствие между парами (a, b) с $a \neq 0$ и парами (r, s), в которых $r \neq s$. Таким образом, для любой данной пары входных значений k и l при равномерном случайном выборе пары (a, b) из $\mathbb{Z}_p^* \times \mathbb{Z}_p$, получаемая в результате пара (r, s) может быть с равной вероятностью любой из пар с отличающимися по модулю p значениями.

Отсюда можно заключить, что вероятность того, что различные ключи k и l приводят к коллизии, равна вероятности того, что $r \equiv s \pmod m$ при случайном выборе отличающихся по модулю p значений r и s. Для данного значения r имеется $p - 1$ возможных значений s. При этом число значений s, таких, что $s \neq r$ и $s \equiv r \pmod m$, не превышает

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \qquad \text{(согласно неравенству (3.6))} \\ &= (p - 1)/m \ . \end{aligned}$$

Вероятность того, что s приводит к коллизии с r при приведении по модулю m, не превышает $((p - 1)/m)/(p - 1) = 1/m$.

Следовательно, для любой пары различных значений $k, l \in \mathbb{Z}_p$

$$\Pr\{h_{ab}(k) = h_{ab}(l)\} \leq 1/m \ ,$$

так что множество хеш-функций $\mathcal{H}_{pm}$ является универсальным. ■

Упражнения

11.3.1

Предположим, что мы выполняем поиск в связанном списке длиной n, в котором каждый элемент содержит ключ k вместе с хеш-значением $h(k)$. Каждый ключ представляет собой длинную символьную строку. Как можно использовать наличие хеш-значения при поиске элемента с заданным ключом?

11.3.2

Предположим, что строка из r символов хешируется в m ячеек путем ее интерпретации как числа, записанного в 128-ричной системе счисления, и использования метода деления. Число m легко представимо в виде 32-битового машинного слова, но представление строки из r символов как целого 128-ричного числа требует много слов. Каким образом можно применить метод деления для вычисления хеш-значения символьной строки с использованием не более чем фиксированного количества дополнительных машинных слов, помимо сам́ой строки?

11.3.3

Рассмотрим версию метода деления, в которой $h(k) = k \bmod m$, где $m = 2^p - 1$, а k – символьная строка, интерпретируемая как целое число в системе счисления с основанием 2^p. Покажите, что если строка x может быть получена из строки y перестановкой символов, то хеш-значения этих строк одинаковы. Приведите пример приложения, в котором это свойство хеш-функции может оказаться крайне нежелательным.

11.3.4

Рассмотрим хеш-таблицу размером $m = 1000$ и соответствующую хеш-функцию $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$, где $A = (\sqrt{5} - 1)/2$. Вычислите номера ячеек, в которые хешируются ключи 61, 62, 63, 64 и 65.

11.3.5 ★

Определим семейство хеш-функций $\mathcal{H}$, отображающих конечное множество U на конечное множество B, как ***ϵ-универсальное***, если для всех пар различных элементов k и l из U

$$\Pr\{h(k) = h(l)\} \le \epsilon ,$$

где вероятность вычисляется для случайного выбора хеш-функции h из множества $\mathcal{H}$. Покажите, что ϵ-универсальное семейство хеш-функций должно обладать тем свойством, что

$$\epsilon \ge \frac{1}{|B|} - \frac{1}{|U|} .$$

11.3.6 ★

Пусть U — множество n-кортежей значений, выбираемых из $\mathbb{Z}_p$, и пусть $B = \mathbb{Z}_p$, где p — простое число. Определим хеш-функцию $h_b : U \to B$ для $b \in \mathbb{Z}_p$

и входного кортежа $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ из U следующим образом:

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \left(\sum_{j=0}^{n-1} a_j b^j \right) \bmod p ,$$

и пусть $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Докажите, что $\mathcal{H}$ является $((n-1)/p)$-универсальным множеством в соответствии с определением, данным в упр. 11.3.5. (*Указание:* см. упр. 31.4.4.)

11.4. Открытая адресация

При использовании метода ***открытой адресации*** все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо значение NIL. При поиске элемента мы систематически проверяем ячейки таблицы до тех пор, пока не найдем искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, в методе открытой адресации хеш-таблица может оказаться заполненной и сделать невозможной вставку новых элементов; одним из следствий этого является то, что коэффициент заполнения α не может превышать 1.

Конечно, при хешировании с разрешением коллизий методом цепочек можно использовать свободные места в хеш-таблице для хранения связанных списков (см. упр. 11.2.4), но преимущество открытой адресации заключается в том, что она позволяет полностью отказаться от указателей. Вместо того чтобы следовать по указателям, мы *вычисляем* последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

Для выполнения вставки при открытой адресации мы последовательно проверяем, или ***исследуем*** (probe), ячейки хеш-таблицы до тех пор, пока не находим пустую ячейку, в которую помещаем вставляемый ключ. Вместо фиксированного порядка исследования ячеек $0, 1, \ldots, m-1$ (для чего требуется $\Theta(n)$ времени), последовательность исследуемых ячеек *зависит от вставляемого в таблицу ключа*. Для определения исследуемых ячеек мы расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0). В результате хеш-функция становится следующей:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} .$$

В методе открытой адресации требуется, чтобы для каждого ключа k ***последовательность исследований***

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

представляла собой перестановку множества $\langle 0, 1, \ldots, m-1 \rangle$, чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы. В приведенном далее псевдокоде предполагается, что элементы в таблице T представляют собой ключи без сопутствующей информации; ключ k тождествен элементу, содержащему ключ k. Каждая ячейка содержит либо ключ, либо значение NIL. Процедура HASH-INSERT получает в качестве входных данных хеш-таблицу T и ключ k. Она либо возвращает номер ячейки, в которую записывается ключ k, либо сообщает об ошибке, связанной с заполненностью таблицы.

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "переполнение хеш-таблицы"
```

Алгоритм поиска ключа k исследует ту же последовательность ячеек, что и алгоритм вставки ключа k. Таким образом, если при поиске встречается пустая ячейка, поиск завершается неудачей, поскольку ключ k должен был бы быть вставлен в эту ячейку в последовательности исследований, и никак не позже нее. (Предполагается, что удалений из хеш-таблицы не было.) Процедура HASH-SEARCH получает в качестве входных параметров хеш-таблицу T и ключ k и возвращает номер ячейки, которая содержит ключ k (или значение NIL, если ключ в хеш-таблице не обнаружен).

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL или i == m
return NIL
```

Процедура удаления из хеш-таблицы с открытой адресацией достаточно сложна. При удалении ключа из ячейки i мы не можем просто пометить ее пустым

значением NIL. Поступив так, мы можем сделать невозможным выборку ключа k, в процессе вставки которого исследовалась и оказалась занятой ячейка i. Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением DELETED вместо NIL. При этом мы должны слегка изменить процедуру HASH-INSERT, чтобы она рассматривала такую ячейку как пустую и могла вставить в нее новый ключ. В процедуре HASH-SEARCH никакие изменения не требуются, поскольку мы просто пропускаем такие ячейки при поиске и исследуем следующие ячейки в последовательности. Однако при использовании специального значения DELETED время поиска перестает зависеть от коэффициента заполнения α, и по этой причине, как правило, при необходимости удалений из хеш-таблицы в качестве метода разрешения коллизий выбирается метод цепочек.

В ходе дальнейшего анализа мы будем исходить из предположения ***равномерного хеширования***, т.е. мы предполагаем, что для каждого ключа в качестве последовательности исследований равновероятны все $m!$ перестановок множества $\langle 0, 1, \ldots, m-1 \rangle$. Равномерное хеширование представляет собой обобщение определенного ранее простого равномерного хеширования, заключающееся в том, что теперь хеш-функция дает не одно значение, а целую последовательность исследований. Реализация истинно равномерного хеширования достаточно трудна, однако на практике используются подходящие аппроксимации (такие, например, как определенное ниже двойное хеширование).

Мы рассмотрим три основных метода вычисления последовательности исследований для открытой адресации: линейное исследование, квадратичное исследование и двойное хеширование. Эти методы гарантируют, что $\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$ является перестановкой для каждого ключа k. Однако эти методы не удовлетворяют предположению о равномерном хешировании, так как ни один из них не в состоянии сгенерировать более m^2 различных последовательностей исследований (вместо $m!$, требующихся для равномерного хеширования). Наибольшее количество последовательностей исследований и, как и следовало ожидать, наилучшие результаты дает двойное хеширование.

Линейное исследование

Пусть задана обычная хеш-функция $h' : U \to \{0, 1, \ldots, m-1\}$, которую мы будем в дальнейшем именовать ***вспомогательной хеш-функцией*** (auxiliary hash function). Метод ***линейного исследования*** для вычисления последовательности исследований использует хеш-функцию

$$h(k,i) = (h'(k) + i) \bmod m$$

для $i = 0, 1, \ldots, m-1$. Для данного ключа k первой исследуемой ячейкой является $T[h'(k)]$, т.е. ячейка, которую дает вспомогательная хеш-функция. Далее мы исследуем ячейку $T[h'(k) + 1]$ и последовательно все до ячейки $T[m-1]$, после чего переходим в начало таблицы и последовательно исследуем ячейки $T[0], T[1], \ldots$, пока не дойдем до ячейки $T[h'(k) - 1]$. Поскольку начальная исследуемая ячейка однозначно определяет всю последовательность исследований целиком, всего имеется m различных последовательностей.

Линейное исследование легко реализуется, однако при этом возникает проблема ***первичной кластеризации***, связанной с созданием длинных последовательностей занятых ячеек, что, само собой разумеется, увеличивает среднее время поиска. Кластеры возникают в связи с тем, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i+1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию ко все большему удлинению, что приводит к увеличению среднего времени поиска.

Квадратичное исследование

Квадратичное исследование использует хеш-функцию вида

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \ , \tag{11.5}$$

где h' — вспомогательная хеш-функция, c_1 и c_2 — положительные вспомогательные константы, а $i = 0, 1, \ldots, m-1$. Начальная исследуемая ячейка — $T[h'(k)]$; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i. Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1, c_2 и m (в задаче 11.3 показан один из путей выбора этих параметров). Кроме того, если два ключа имеют одну и ту же начальную позицию исследования, то одинаковы и последовательности исследования в целом, так как из $h(k_1, 0) = h(k_2, 0)$ вытекает $h(k_1, i) = h(k_2, i)$. Это свойство приводит к более мягкой ***вторичной кластеризации***. Как и в случае линейного исследования, начальная ячейка определяет всю последовательность, поэтому всего используется m различных последовательностей исследования.

Двойное хеширование

Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок. ***Двойное хеширование*** использует хеш-функцию вида

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m \ ,$$

где и h_1, и h_2 — вспомогательные хеш-функции. Начальное исследование выполняется в позиции $T[h_1(k)]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю m. Следовательно, в отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа k по двум параметрам — в плане выбора начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа. На рис. 11.5 приведен пример вставки при двойном хешировании.

Для того чтобы последовательность исследования могла охватить всю таблицу, значение $h_2(k)$ должно быть взаимно простым с размером хеш-таблицы m

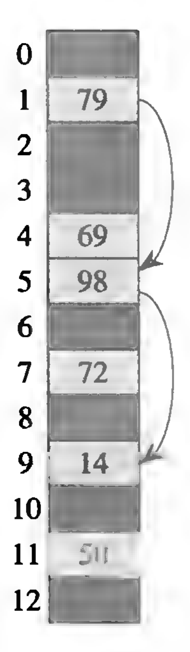


Рис. 11.5. Вставка при двойном хешировании. Здесь показана хеш-таблица размером 13 ячеек, в которой используются хеш-функции $h_1(k) = k \bmod 13$ и $h_2(k) = 1 + (k \bmod 11)$. Так как $14 \equiv 1 \pmod{13}$ и $14 \equiv 3 \pmod{11}$, ключ 14 вставляется в пустую ячейку 9, после того как при исследовании ячеек 1 и 5 выясняется, что эти ячейки заняты.

(см. упр. 11.4.4). Удобный способ обеспечить выполнение этого условия состоит в выборе числа m, равного степени 2, и разработке хеш-функции h_2 таким образом, чтобы она возвращала только нечетные значения. Еще один способ состоит в использовании в качестве m простого числа и построении хеш-функции h_2, такой, чтобы она всегда возвращала натуральные числа, меньшие m. Например, можно выбрать простое m и хеш-функции

$$h_1(k) = k \bmod m ,$$
$$h_2(k) = 1 + (k \bmod m') ,$$

где m' должно быть немного меньше m (скажем, $m - 1$). Например, если $k = 123456$, $m = 701$, а $m' = 700$, мы имеем $h_1(k) = 80$ и $h_2(k) = 257$, так что первой исследуемой будет ячейка в 80-й позиции, а затем будет исследоваться каждая 257-я (по модулю m) ячейка, пока не будет обнаружена пустая ячейка или пока не будут исследованы все ячейки таблицы.

Когда m простое или представляет собой степень 2, двойное хеширование превосходит линейное или квадратичное исследования в смысле количества $\Theta(m^2)$ последовательностей исследований, в то время как у упомянутых методов это количество равно $\Theta(m)$, поскольку каждая возможная пара $(h_1(k), h_2(k))$ дает отличную от других последовательность исследований. В результате для таких значений m производительность двойного хеширования достаточно близка к производительности "идеальной" схемы равномерного хеширования.

Хотя в принципе для двойного хеширования могут использоваться значения m, отличные от простых и степеней 2, на практике при этом становится труднее эффективно генерировать $h_2(k)$ так, чтобы гарантировать взаимную простоту с m,

в частности из-за того, что относительная плотность $\phi(m)/m$ таких чисел может быть малой (см. уравнение (31.24)).

Анализ хеширования с открытой адресацией

Анализ открытой адресации, как и анализ метода цепочек, выполняется с использованием коэффициента заполнения $\alpha = n/m$. Само собой разумеется, при использовании открытой адресации может быть не более одного элемента на ячейку таблицы, так что $n \leq m$ и, следовательно, $\alpha \leq 1$.

Будем считать, что используется равномерное хеширование. При такой идеализированной схеме последовательность исследований $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$, используемая для вставки или поиска каждого ключа k, с равной вероятностью является одной из возможных перестановок $\langle 0, 1, \ldots, m-1\rangle$. Разумеется, с каждым конкретным ключом связана единственная фиксированная последовательность исследований, так что при рассмотрении распределения вероятностей ключей и хеш-функций все последовательности исследований оказываются равновероятными.

Мы проанализируем математическое ожидание количества исследований для хеширования с открытой адресацией в предположении равномерного хеширования и начнем с анализа количества исследований в случае неудачного поиска.

Теорема 11.6

Математическое ожидание количества исследований при неудачном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha = n/m < 1$ в предположении равномерного хеширования не превышает $1/(1-\alpha)$.

Доказательство. При неудачном поиске каждая последовательность исследований завершается пустой ячейкой. Определим случайную величину X как количество исследований, выполненных при неудачном поиске, и определим также события A_i ($i = 1, 2, \ldots$), заключающиеся в том, что было выполнено i-е исследование, и оно пришлось на занятую ячейку. Тогда событие $\{X \geq i\}$ представляет собой пересечение событий $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. Ограничим вероятность $\Pr\{X \geq i\}$ путем ограничения вероятности $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. В соответствии с упр. В.2.5

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} .$$

Поскольку всего имеется n элементов и m ячеек, $\Pr\{A_1\} = n/m$. Вероятность того, что будет выполнено j-е исследование ($j > 1$) и что оно будет проведено над заполненной ячейкой (при этом первые $j-1$ исследований проведены над заполненными ячейками), равна $(n-j+1)/(m-j+1)$. Эта вероятность определяется следующим образом: мы должны проверить один из оставшихся $(n-(j-1))$ элементов в одной из оставшихся к этому времени $(m-(j-1))$ неисследованных ячеек. В соответствии с предположением о равномерном хешировании искомая вероятность равна отношению этих величин. Воспользовавшись тем фактом, что

из $n < m$ для всех $0 \le j < m$ следует соотношение $(n-j)/(m-j) \le n/m$, для всех $1 \le i \le m$ получаем

$$\begin{aligned}\Pr\{X \ge i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\le \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} .\end{aligned}$$

Теперь воспользуемся уравнением (В.25) для получения границы ожидаемого количества исследований:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \ge i\} \\ &\le \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} .\end{aligned}$$

■

Полученная граница $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ имеет интуитивную интерпретацию. Одно исследование выполняется всегда. С вероятностью, приблизительно равной α, первое исследование проводится над заполненной ячейкой, и требуется выполнение второго исследования. С вероятностью, приблизительно равной α^2, две первые ячейки оказываются заполненными, и требуется проведение третьего исследования, и т.д.

Если α — константа, то теорема 11.6 предсказывает, что неудачный поиск выполняется за время $O(1)$. Например, если хеш-таблица заполнена наполовину, то среднее количество исследований при неудачном поиске не превышает $1/(1-.5) = 2$. При заполненности хеш-таблицы на 90% среднее количество исследований не превышает $1/(1-.9) = 10$.

Теорема 11.6 практически непосредственно дает оценку производительности процедуры HASH-INSERT.

Следствие 11.7

Вставка элемента в хеш-таблицу с открытой адресацией и коэффициентом заполнения α в предположении равномерного хеширования требует в среднем не более $1/(1-\alpha)$ исследований.

Доказательство. Элемент может быть вставлен в хеш-таблицу только в том случае, если в ней есть свободное место, так что $\alpha < 1$. Вставка ключа требует проведения неудачного поиска, за которым следует размещение ключа в найден-

ной пустой ячейке. Следовательно, математическое ожидание количества исследований не превышает $1/(1-\alpha)$. ■

Вычисление математического ожидания количества исследований при успешном поиске требует немного больше усилий.

Теорема 11.8

Математическое ожидание количества исследований при удачном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha < 1$, в предположении равномерного хеширования и равновероятного поиска любого из ключей, не превышает

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}.$$

Доказательство. Поиск ключа k выполняется той же последовательностью исследований, что и его вставка. В соответствии со следствием 11.7, если k был $(i+1)$-м ключом, вставленным в хеш-таблицу, то математическое ожидание количества проб при поиске k не превышает $1/(1-i/m) = m/(m-i)$. Усреднение по всем n ключам в хеш-таблице дает нам среднее количество исследований при удачном поиске:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \qquad \text{(согласно неравенству (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}.
\end{aligned}$$
■

В наполовину заполненной хеш-таблице ожидаемое количество исследований при удачном поиске оказывается меньше 1.387. Если хеш-таблица заполнена на 90%, ожидаемое количество исследований меньше 2.559.

Упражнения

11.4.1

Рассмотрите вставку ключей 10, 22, 31, 4, 15, 28, 17, 88, 59 в хеш-таблицу длиной $m = 11$ с открытой адресацией и вспомогательной хеш-функцией $h'(k) = k$. Проиллюстрируйте результат вставки приведенного списка ключей при использовании линейного исследования, квадратичного исследования с $c_1 = 1$ и $c_2 = 3$ и двойного хеширования с $h_1(k) = k$ и $h_2(k) = 1 + (k \bmod (m-1))$.

11.4.2

Запишите псевдокод процедуры HASH-DELETE, описанной в тексте, и модифицируйте процедуру HASH-INSERT так, чтобы она могла обрабатывать специальное значение DELETED.

11.4.3

Рассмотрим хеш-таблицу с открытой адресацией при условии равномерного хеширования. Найдите верхнюю границу ожидаемого количества исследований при неудачном поиске и ожидаемого количества исследований при удачном поиске, когда коэффициент заполнения равен $3/4$ и когда он равен $7/8$.

11.4.4 ★

Предположим, что для разрешения коллизий мы используем двойное хеширование, т.е. хеш-функцию $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Покажите, что если m и $h_2(k)$ для некоторого ключа k имеют наибольший общий делитель $d \geq 1$, то неудачный поиск ключа k проверяет $(1/d)$-ю часть хеш-таблицы до того, как возвращается в ячейку $h_1(k)$. Таким образом, когда $d = 1$, т.е. m и $h_2(k)$ — взаимно простые числа, поиск может исследовать всю хеш-таблицу. (*Указание:* см. главу 31.)

11.4.5 ★

Рассмотрим хеш-таблицу с открытой адресацией и коэффициентом заполнения α. Найдите ненулевое значение α, при котором математическое ожидание количества исследований в случае неудачного поиска в два раза превышает математическое ожидание количества исследований в случае удачного поиска. Воспользуйтесь для решения поставленной задачи границами, приведенными в теоремах 11.6 и 11.8.

★ 11.5. Идеальное хеширование

Хотя чаще всего хеширование используется из-за превосходной средней производительности, возможна ситуация, когда можно обеспечить превосходную производительность хеширования в *наихудшем* случае. Такой ситуацией является ***статическое*** множество ключей, т.е. после того как все ключи сохранены в таблице, их множество никогда не изменяется. Ряд приложений в силу своей природы работает со статическими множествами ключей. В качестве примера можно привести множество зарезервированных слов языка программирования или множество имен файлов на компакт-диске. ***Идеальным хешированием*** мы называем методику, которая выполняет поиск за $O(1)$ обращений к памяти в наихудшем случае.

Для создания схемы идеального хеширования мы используем двухуровневую схему хеширования с универсальным хешированием на каждом уровне (см. рис. 11.6).

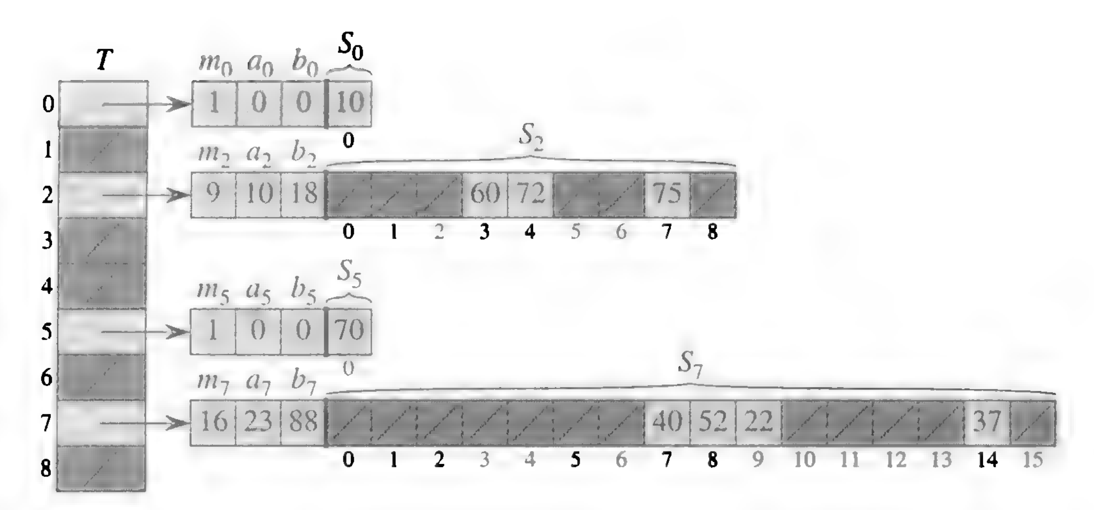


Рис. 11.6. Использование идеального хеширования для хранения множества $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. Внешняя хеш-функция имеет вид $h(k) = ((ak + b) \bmod p) \bmod m$, где $a = 3$, $b = 42$, $p = 101$ и $m = 9$. Например, $h(75) = 2$, так что ключ 75 хешируется в ячейку 2 таблицы T. Вторичная хеш-таблица S_j хранит все ключи, хешированные в ячейку j. Размер каждой таблицы S_j равен $m_j = n_j^2$, и с ней связана хеш-функция $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Поскольку $h_2(75) = 7$, ключ 75 хранится в ячейке 7 вторичной хеш-таблицы S_2. Ни в одной из вторичных таблиц нет ни одной коллизии, так что время поиска в худшем случае равно константе.

Первый уровень по сути тот же, что и в случае хеширования с цепочками: n ключей хешируются в m ячеек с использованием хеш-функции h, тщательно выбранной из семейства универсальных хеш-функций.

Однако вместо того, чтобы создавать список ключей, хешированных в ячейку j, мы используем маленькую ***вторичную хеш-таблицу*** S_j со связанной с ней хеш-функцией h_j. Путем аккуратного выбора хеш-функции h_j мы можем гарантировать отсутствие коллизий на втором уровне.

Чтобы гарантировать отсутствие коллизий на втором уровне, требуется, чтобы размер m_j хеш-таблицы S_j был равен квадрату числа n_j ключей, хешированных в ячейку j. Такая квадратичная зависимость m_j от n_j может показаться чрезмерно расточительной, однако далее мы покажем, что при должном выборе хеш-функции первого уровня ожидаемое количество требуемой для хеш-таблицы памяти можно ограничить значением $O(n)$.

Мы выбираем хеш-функцию из универсальных классов хеш-функций из раздела 11.3.3. Хеш-функция первого уровня выбирается из класса $\mathcal{H}_{pm}$, где, как и в разделе 11.3.3, p является простым числом, превышающим значение любого из ключей. Ключи, хешированные в ячейку j, затем повторно хешируются во вторичную хеш-таблицу S_j размером m_j с использованием хеш-функции h_j, выбранной из класса $\mathcal{H}_{p,m_j}$[2].

Работа будет выполнена в два этапа. Сначала мы выясним, как гарантировать отсутствие коллизий во вторичной таблице. Затем мы покажем, что общее ожидаемое количество памяти, необходимой для первичной и всех вторичных хеш-таблиц, равно $O(n)$.

---

[2]При $n_j = m_j = 1$ для ячейки j хеш-функция не нужна; при выборе хеш-функции $h_{ab}(k) = ((ak + b) \bmod p) \bmod m_j$ для такой ячейки мы просто выбираем $a = b = 0$.

Теорема 11.9

Предположим, что n ключей сохраняются в хеш-таблице размером $m = n^2$ с использованием хеш-функции h, случайно выбранной из универсального класса хеш-функций. Тогда вероятность возникновения коллизий оказывается меньше $1/2$.

Доказательство. Всего имеется $\binom{n}{2}$ пар ключей, которые могут вызвать коллизию. Если хеш-функция выбрана случайным образом из универсального семейства хеш-функций $\mathcal{H}$, то для каждой пары вероятность возникновения коллизии равна $1/m$. Пусть X — случайная величина, которая подсчитывает количество коллизий. Если $m = n^2$, то математическое ожидание числа коллизий равно

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2 \,. \end{aligned}$$

(Обратите внимание на схожесть данного анализа с анализом парадокса дней рождения из раздела 5.4.1.) Применение неравенства Маркова (В.30), $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, при $t = 1$ завершает доказательство. ■

В ситуации, описанной в теореме 11.9, когда $m = n^2$, произвольно выбранная из множества $\mathcal{H}$ хеш-функция с большей вероятностью *не* приведет к коллизиям, чем приведет к ним. Для заданного множества K, содержащего n ключей (напомним, что K — статическое множество), найти хеш-функцию h, не дающую коллизий, можно после нескольких случайных попыток.

Однако если значение n велико, таблица размером $m = n^2$ оказывается слишком большой и приводит к ненужному перерасходу памяти. Поэтому мы принимаем двухуровневую схему хеширования и используем подход из теоремы 11.9 только для хеширования записей в пределах каждой ячейки. Внешняя хеш-функция h первого уровня используется для хеширования ключей в $m = n$ ячеек. Затем, если в ячейку j хешировано n_j ключей, для того чтобы обеспечить отсутствие коллизий и поиск за константное время, используется вторичная хеш-таблица S_j размером $m_j = n_j^2$.

Вернемся к вопросу необходимого для описанной схемы количества памяти. Поскольку размер j-й вторичной хеш-таблицы m_j растет с ростом n_j квадратично, возникает риск, что в целом потребуется очень большое количество памяти.

Если хеш-таблица первого уровня имеет размер $m = n$, то, естественно, нам потребуется количество памяти, равное $O(n)$, для первичной хеш-таблицы, а также для хранения размеров m_j вторичных хеш-таблиц и параметров a_j и b_j, определяющих вторичные хеш-функции h_j, выбираемые из класса $\mathcal{H}_{p,m_j}$ из раздела 11.3.3 (за исключением случая, когда $n_j = 1$; в этом случае мы просто принимаем $a = b = 0$). Следующая теорема и следствие из нее позволяют нам вычислить границу суммарного размера всех вторичных таблиц. Второе следствие

ограничивает вероятность того, что объединенный размер всех вторичных хеш-таблиц надлинеен (в действительности вероятность того, что он равен или превышает $4n$).

Теорема 11.10

Предположим, что мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций. Тогда мы имеем

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n ,$$

где n_j — количество ключей, хешированных в ячейку j.

Доказательство. Начнем со следующего тождества, которое справедливо для любого неотрицательного целого a:

$$a^2 = a + 2\binom{a}{2} . \tag{11.6}$$

Мы имеем

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] & \\
&= \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(согласно уравнению (11.6))} \\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(из линейности математического ожидания)} \\
&= \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(согласно уравнению (11.1))} \\
&= n + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(так как } n \text{ не является случайной величиной)} .
\end{aligned}$$

Для того чтобы вычислить сумму $\sum_{j=0}^{m-1}\binom{n_j}{2}$, заметим, что это просто общее количество коллизий. В соответствии со свойством универсального хеширования математическое ожидание значения этой суммы не превышает

$$\begin{aligned}
\binom{n}{2}\frac{1}{m} &= \frac{n(n-1)}{2m} \\
&= \frac{n-1}{2} ,
\end{aligned}$$

поскольку $m = n$. Таким образом,

$$\begin{aligned} \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &\le n + 2\,\frac{n-1}{2} \\ &= 2n - 1 \\ &< 2n\ . \end{aligned}$$ ■

Следствие 11.11

Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$, $j = 0, 1, \ldots, m-1$, то математическое ожидание количества необходимой для вторичных хеш-таблиц в схеме идеального хеширования памяти не превышает величины $2n$.

Доказательство. Поскольку $m_j = n_j^2$ для $j = 0, 1, \ldots, m-1$, теорема 11.10 дает

$$\begin{aligned} \mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\ &< 2n\ , \end{aligned} \qquad (11.7)$$

что и требовалось доказать. ■

Следствие 11.12

Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального класса хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$, $j = 0, 1, \ldots, m-1$, то вероятность того, что общее количество необходимой для вторичных хеш-таблиц памяти не менее $4n$, меньше, чем $1/2$.

Доказательство. Вновь применим неравенство Маркова (В.30), $\Pr\{X \ge t\} \le \mathrm{E}[X]/t$, на этот раз — к неравенству (11.7), с $X = \sum_{j=0}^{m-1} m_j$ и $t = 4n$:

$$\begin{aligned} \Pr\left\{\sum_{j=0}^{m-1} m_j \ge 4n\right\} &\le \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2\ . \end{aligned}$$ ■

Из следствия 11.12 видно, что если проверить несколько случайным образом выбранных из универсального семейства хеш-функций, то достаточно быстро бу-

дет найдена хеш-функция, обеспечивающая умеренные требования к количеству памяти.

Упражнения

11.5.1 ★

Предположим, что мы вставляем n ключей в хеш-таблицу размером m с использованием открытой адресации и равномерного хеширования. Обозначим через $p(n, m)$ вероятность отсутствия коллизий. Покажите, что $p(n, m) \leq e^{-n(n-1)/2m}$. (*Указание:* см. уравнение (3.12).) Докажите, что при n, превосходящем $\sqrt{m}$, вероятность избежать коллизий быстро уменьшается до нуля.

Задачи

11.1. Наибольшее число исследований при хешировании

Предположим, что мы используем хеш-таблицу размером m для хранения $n \leq m/2$ элементов.

а. Покажите, что в предположении равномерного хеширования вероятность того, что для любого $i = 1, 2, \ldots, n$ для вставки i-го ключа требуется более k исследований, не превышает 2^{-k}.

б. Покажите, что для $i = 1, 2, \ldots, n$ вероятность того, что вставка i-го ключа требует выполнения более $2 \lg n$ исследований, не превышает $1/n^2$.

Пусть X_i — случайная величина, обозначающая количество исследований, необходимое при вставке i-го ключа. В подзадаче (б) нужно было показать, что $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Обозначим через X максимальное количество исследований, необходимое при любой из n вставок: $X = \max_{1 \leq i \leq n} X_i$.

в. Покажите, что $\Pr\{X > 2 \lg n\} = O(1/n)$.

г. Покажите, что математическое ожидание $\mathrm{E}[X]$ длины наибольшей последовательности исследований равно $O(\lg n)$.

11.2. Длины цепочек при хешировании

Предположим, что имеется хеш-таблица с n ячейками и разрешением коллизий методом цепочек. Пусть в таблицу вставлено n ключей. Вероятность каждого ключа попасть в любую из ячеек одинакова для всех ключей и всех ячеек. Пусть M — максимальное количество ключей в ячейке после того, как все ключи вставлены в таблицу. Ваша задача — доказать, что математическое ожидание $\mathrm{E}[M]$ имеет верхнюю границу $O(\lg n / \lg \lg n)$.

а. Докажите, что вероятность Q_k того, что в определенной ячейке находится ровно k ключей, равна

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

б. Пусть P_k — вероятность того, что $M = k$, т.е. вероятность того, что ячейка, содержащая наибольшее количество ключей, содержит их ровно k. Покажите, что $P_k \le nQ_k$.

в. Воспользуйтесь приближением Стирлинга (3.18), чтобы показать, что $Q_k < e^k/k^k$.

г. Покажите, что существует константа $c > 1$, такая, что $Q_{k_0} < 1/n^3$ для $k_0 = c \lg n / \lg\lg n$. Выведите отсюда, что $P_k < 1/n^2$ для $k \ge k_0 = c \lg n / \lg\lg n$.

д. Докажите, что

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg\lg n}\right\} \cdot \frac{c \lg n}{\lg\lg n} .$$

Сделайте вывод, что $\mathrm{E}[M] = O(\lg n / \lg\lg n)$.

11.3. Квадратичное исследование

Предположим, что есть ключ k, который нужно найти в хеш-таблице с ячейками $0, 1, \ldots, m-1$, и пусть есть хеш-функция h, отображающая пространство ключей в множество $\{0, 1, \ldots, m-1\}$. Схема поиска выглядит следующим образом.

1. Вычисляем значение $j = h(k)$ и присваиваем $i = 0$.
2. Исследуем ячейку j на соответствие искомому ключу k. Если ключ найден или если ячейка пуста, поиск прекращается.
3. Присваиваем $i = i + 1$. Если i равно m, поиск прекращается. В противном случае присваиваем $j = (i + j) \bmod m$ и возвращаемся к шагу 2.

Предположим, что m является степенью 2.

а. Покажите, что описанная схема представляет собой частный случай общей схемы "квадратичного исследования" с соответствующими константами c_1 и c_2 в уравнении (11.5).

б. Докажите, что этот алгоритм в наихудшем случае исследует каждую ячейку таблицы.

11.4. Хеширование и аутентификация

Пусть $\mathcal{H}$ представляет собой класс хеш-функций, в котором каждая хеш-функция $h \in \mathcal{H}$ отображает совокупность ключей U на множество $\{0, 1, \ldots, m-1\}$.

Мы говорим, что $\mathcal{H}$ является ***k*-универсальным**, если для каждой фиксированной последовательности из k различных ключей $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)}\rangle$ и случайным образом выбранной из $\mathcal{H}$ хеш-функции h последовательность $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)})\rangle$ с равной вероятностью является любой из m^k последовательностей длиной k из элементов множества $\{0, 1, \ldots, m-1\}$.

а. Покажите, что если семейство $\mathcal{H}$ хеш-функций 2-универсально, то оно является универсальным.

б. Предположим, что совокупность U представляет собой множество n-кортежей значений, выбранных из $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, где p — простое число. Рассмотрим элемент $x = \langle x_0, x_1, \ldots, x_{n-1}\rangle \in U$. Для любого n-кортежа $a = \langle a_0, a_1, \ldots, a_{n-1}\rangle \in U$ определим хеш-функцию h_a следующим образом:

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p \; .$$

Пусть $\mathcal{H} = \{h_a\}$. Покажите, что $\mathcal{H}$ универсальна, но не 2-универсальна. (*Указание:* найдите ключ, для которого все хеш-функции из $\mathcal{H}$ дают одно и то же значение.)

в. Предположим, что мы слегка изменили $\mathcal{H}$ из п. (б): для любого $a \in U$ и для любого $b \in \mathbb{Z}_p$ определим

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b\right) \bmod p$$

и $\mathcal{H}' = \{h'_{ab}\}$. Докажите, что класс $\mathcal{H}'$ является 2-универсальным. (*Указание:* рассмотрите фиксированные n-кортежи $x \in U$ и $y \in U$, с $x_i \neq y_i$ для некоторого i. Что произойдет с $h'_{ab}(x)$ и $h'_{ab}(y)$ при пробегании a_i и b всех значений из $\mathbb{Z}_p$?)

г. Два друга по секрету договорились о хеш-функции h из 2-универсального семейства хеш-функций $\mathcal{H}$. Каждая функция $h \in \mathcal{H}$ отображает совокупность ключей U на $\mathbb{Z}_p$, где p — простое число. Один из друзей шлет сообщение m другому через Интернет. Он подписывает это сообщение, пересылая одновременно с ним дескриптор $t = h(m)$, а его друг проверяет, удовлетворяет ли полученная им пара (m, t) условию $t = h(m)$. Предположим, что злоумышленник перехватывает сообщение (m, t) и пытается обмануть получателя, подсунув ему собственное сообщение (m', t'). Покажите, что вероятность того, что злоумышленник сможет успешно совершить подмену, не превышает $1/p$, независимо от того, какими вычислительными мощностями он обладает, даже если он заранее знает семейство используемых хеш-функций $\mathcal{H}$.

Заключительные замечания

Алгоритмы хеширования прекрасно изложены в книгах Кнута (Knuth) [210][3] и Гонне (Gonnet) [144]. Согласно Кнуту хеш-таблицы и метод цепочек были изобретены Г.П. Луном (H.P. Luhn) в 1953 году. Примерно тогда же Ж.М. Амдаль (G.M. Amdahl) предложил идею открытой адресации.

Универсальные множества хеш-функций были предложены Картером (Carter) и Вегманом (Wegman) в 1979 году [57].

Схему идеального хеширования для статических множеств, представленную в разделе 11.5, разработали Фредман (Fredman), Комлёс (Komlós) и Семереди (Szemerédi) [111]. Расширение этого метода для динамических множеств, обрабатывающее вставки и удаления за амортизированное ожидаемое время $O(1)$, предложено Дицфельбингером (Dietzfelbinger) и др. [85].

[3]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 12. Бинарные деревья поиска

Деревья поиска представляют собой структуры данных, которые поддерживают многие операции с динамическими множествами, включая SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT и DELETE. Таким образом, дерево поиска может использоваться и как словарь, и как очередь с приоритетами.

Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время $\Theta(\lg n)$ в наихудшем случае. Однако, если дерево представляет собой линейную цепочку из n узлов, те же операции выполняются в наихудшем случае за время $\Theta(n)$. Как будет показано в разделе 12.4, математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\lg n)$, так что все основные операции над динамическим множеством в таком дереве выполняются в среднем за время $\Theta(\lg n)$.

На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. В главе 13 будет представлена одна из таких версий, а именно — красно-черные деревья, высота которых составляет $O(\lg n)$. В главе 18 вы познакомитесь с В-деревьями, которые особенно хорошо подходят для баз данных, хранящихся во вторичной памяти с произвольным доступом (на дисках).

После знакомства с основными свойствами деревьев поиска в последующих разделах главы будет показано, как осуществляется обход дерева поиска для вывода его элементов в отсортированном порядке, как выполняется поиск минимального и максимального элементов, а также предшествующего данному элементу и следующего за ним, как вставлять элементы в дерево поиска и удалять их оттуда. Основные математические свойства деревьев описаны в приложении Б.

12.1. Что такое бинарное дерево поиска

Как следует из названия, бинарное дерево поиска, в первую очередь, является бинарным деревом, как показано на рис. 12.1. Такое дерево может быть представлено с помощью связанной структуры данных, в которой каждый узел является

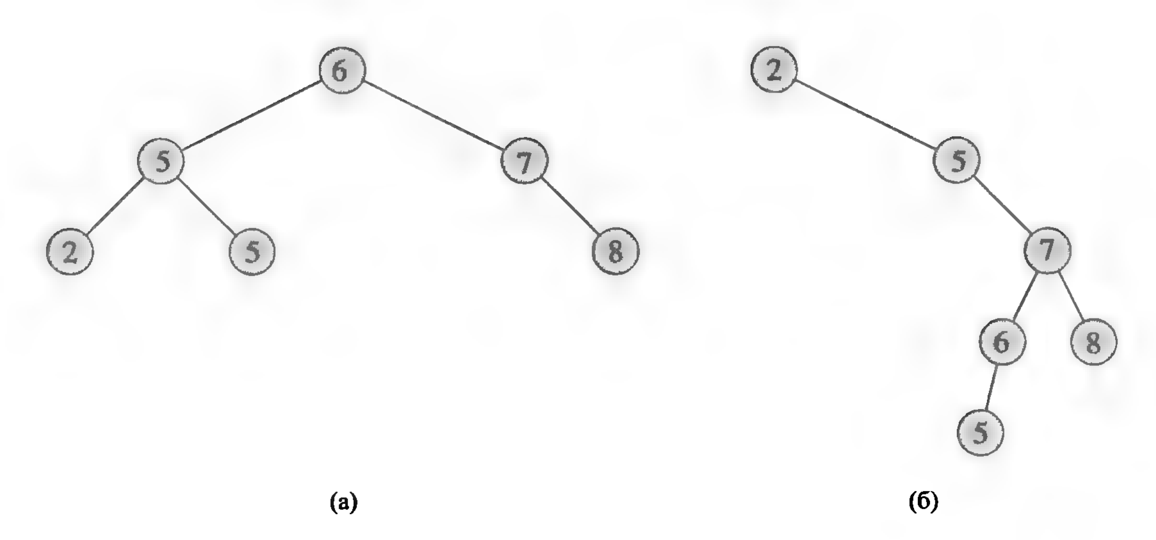


Рис. 12.1. Бинарные деревья поиска. Для любого узла x ключи в левом поддереве x не превышают $x.key$, а ключи в правом поддереве x — не меньше $x.key$. Одно и то же множество значений могут представлять различные бинарные деревья поиска. Время работы в наихудшем случае для большинства операций в дереве поиска пропорционально его высоте. **(а)** Бинарное дерево поиска с 6 узлами высотой 2. **(б)** Менее эффективное бинарное дерево поиска с высотой 4, содержащее те же узлы.

объектом. В дополнение к атрибуту ключа key и сопутствующим данным каждый узел содержит атрибуты $left$, $right$ и p, которые указывают на левый и правый дочерние узлы и на родительский узел соответственно. Если дочерний или родительский узел отсутствуют, соответствующее поле содержит значение NIL. Единственный узел, указатель p которого равен NIL, — это корневой узел дерева.

Ключи в бинарном дереве поиска хранятся таким образом, чтобы в любой момент удовлетворять следующему ***свойству бинарного дерева поиска***.

> Пусть x представляет собой узел бинарного дерева поиска. Если y является узлом в левом поддереве x, то $y.key \leq x.key$. Если y является узлом в правом поддереве x, то $y.key \geq x.key$.

Таким образом, на рис. 12.1,(а) ключом корня является значение 6, ключи 2, 5 и 5 в его левом поддереве не превосходят значение 6, а ключи 7 и 8 в его правом поддереве не меньше 6. То же свойство выполняется для каждого узла дерева. Например, ключ 5 в левом дочернем по отношению к корню узле не меньше ключа 2 в его левом поддереве и не больше ключа 5 в его правом поддереве.

Свойство бинарного дерева поиска позволяет вывести все ключи, находящиеся в дереве, в отсортированном порядке с помощью простого рекурсивного алгоритма, называемого ***центрированным (симметричным) обходом дерева*** (inorder tree walk). Этот алгоритм получил данное название в связи с тем, что ключ в корне поддерева выводится между значениями ключей левого поддерева и правого поддерева. Имеются и другие способы обхода, а именно — ***обход в прямом порядке*** (preorder tree walk), при котором сначала выводится корень, а затем — значения левого и правого поддеревьев, и ***обход в обратном порядке*** (postorder tree walk), когда первыми выводятся значения левого и правого поддеревьев, а уже затем — корня. Центрированный обход бинарного дерева поиска T реализуется вызовом процедуры INORDER-TREE-WALK($T.root$).

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

В качестве примера центрированный обход деревьев, показанных на рис. 12.1, выводит ключи в порядке $2, 5, 5, 6, 7, 8$. Корректность описанного алгоритма следует по индукции непосредственно из свойства бинарного дерева поиска.

Для обхода дерева с n узлами требуется время $\Theta(n)$, поскольку после начального вызова процедура вызывается ровно два раза для каждого узла дерева: один раз – для его левого дочернего узла и один раз – для правого. Приведенная далее теорема дает нам более формальное доказательство линейности времени центрированного обхода дерева.

Теорема 12.1

Если x – корень поддерева с n узлами, то время работы вызова INORDER-TREE-WALK(x) составляет $\Theta(n)$.

Доказательство. Обозначим через $T(n)$ время, необходимое процедуре INORDER-TREE-WALK в случае вызова с параметром, представляющим собой корень дерева с n узлами. Поскольку процедура INORDER-TREE-WALK посещает все n узлов поддерева, мы имеем $T(n) = \Omega(n)$. Остается показать, что $T(n) = O(n)$.

Поскольку INORDER-TREE-WALK требует маленького, фиксированного количества времени для работы с пустым деревом (для выполнения проверки $x \neq \text{NIL}$), мы имеем $T(0) = c$ для некоторой константы $c > 0$.

В случае $n > 0$ будем считать, что процедура INORDER-TREE-WALK вызывается в узле x один раз для левого поддерева с k узлами, а второй – для правого поддерева с $n - k - 1$ узлами. Таким образом, время работы процедуры INORDER-TREE-WALK(x) ограничено $T(n) \leq T(k) + T(n-k-1) + d$ для некоторой константы $d > 0$, которая отражает время, необходимое для выполнения тела процедуры без учета рекурсивных вызовов.

Воспользуемся методом подстановки, чтобы показать, что $T(n) = O(n)$, путем доказательства того, что $T(n) \leq (c+d)n + c$. Для $n = 0$ мы имеем $(c+d) \cdot 0 + c = c = T(0)$. Для $n > 0$ мы имеем

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c\ , \end{aligned}$$

что и завершает доказательство. ■

11 Зак. 3726

Упражнения

12.1.1

Начертите бинарные деревья поиска высотой 2, 3, 4, 5 и 6 для множества ключей $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1.2

В чем заключается отличие свойства бинарного дерева поиска от свойства неубывающей пирамиды (с. 181)? Можно ли использовать свойство неубывающей пирамиды для вывода ключей дерева с n узлами в отсортированном порядке за время $O(n)$? Поясните свой ответ.

12.1.3

Разработайте нерекурсивный алгоритм, осуществляющий обход дерева в симметричном порядке. (*Указание:* имеется простое решение, которое использует вспомогательный стек, и более сложное, но более элегантное решение, которое обходится без стека, но предполагает возможность проверки равенства двух указателей.)

12.1.4

Разработайте рекурсивный алгоритм, который осуществляет прямой и обратный обходы дерева с n узлами за время $\Theta(n)$.

12.1.5

Покажите, что, поскольку сортировка n элементов требует в модели сортировки сравнением в худшем случае времени $\Omega(n \lg n)$, любой основанный на сравнениях алгоритм построения бинарного дерева поиска из произвольного списка, содержащего n элементов, также требует в худшем случае времени $\Omega(n \lg n)$.

12.2. Работа с бинарным деревом поиска

Наиболее частой операцией, выполняемой с бинарным деревом поиска, является поиск в нем определенного ключа. Помимо операции SEARCH, бинарные деревья поиска поддерживают такие запросы, как MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR. В данном разделе мы рассмотрим все эти операции и покажем, что все они могут быть выполнены в бинарном дереве поиска высотой h за время $O(h)$.

Поиск

Для поиска узла с заданным ключом в бинарном дереве поиска используется приведенная ниже процедура TREE-SEARCH, которая получает в качестве параметров указатель на корень бинарного дерева и ключ k, а возвращает указатель на узел с этим ключом (если таковой существует; в противном случае возвращается значение NIL).

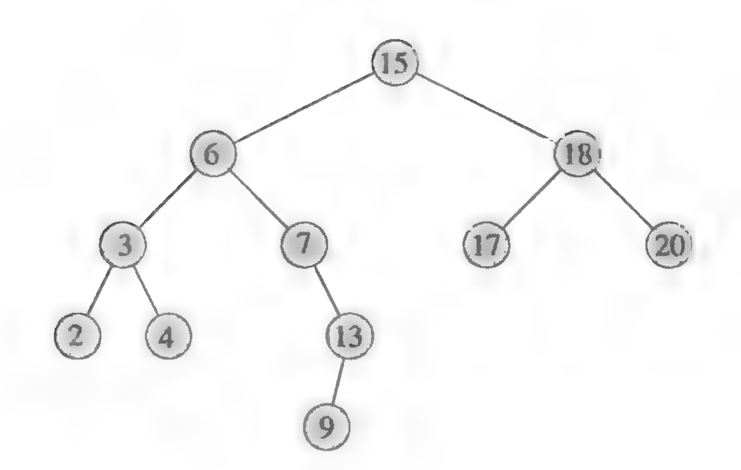


Рис. 12.2. Запросы к бинарному дереву поиска. Для поиска ключа 13 необходимо пройти путь $15 \to 6 \to 7 \to 13$ от корня. Минимальным в дереве является ключ 2, который находится при следовании по указателям *left* от корня. Максимальный ключ 20 достигается при следовании от корня по указателям *right*. Последующим узлом после узла с ключом 15 является узел с ключом 17, поскольку это минимальный ключ в правом поддереве узла 15. Узел с ключом 13 не имеет правого поддерева, так что следующим за ним является наименьший предок, левый наследник которого также является предком данного узла. В нашем случае это узел с ключом 15.

TREE-SEARCH(x, k)

if x == NIL или $k == x.key$
    return x
if $k < x.key$
    return TREE-SEARCH$(x.left, k)$
else return TREE-SEARCH$(x.right, k)$

Процедура поиска начинается с корня дерева и проходит вниз по простому пути по дереву, как показано на рис. 12.2. Для каждого встреченного на пути вниз узла x его ключ $x.key$ сравнивается с переданным в качестве параметра ключом k. Если ключи одинаковы, поиск завершается. Если k меньше $x.key$, поиск продолжается в левом поддереве x, так как из свойства бинарного дерева поиска понятно, что искомый ключ не может находиться в правом поддереве. Аналогично, если k больше $x.key$, поиск продолжается в правом поддереве x. Узлы, которые мы посещаем при рекурсивном поиске, образуют простой нисходящий путь от корня дерева, так что время работы процедуры TREE-SEARCH равно $O(h)$, где h — высота дерева.

Ту же процедуру можно записать итеративно, "развернув" оконечную рекурсию в цикл **while**. На большинстве компьютеров такая версия оказывается более эффективной.

ITERATIVE-TREE-SEARCH(x, k)

while $x \neq$ NIL и $k \neq x.key$
    if $k < x.key$
        $x = x.left$
    else $x = x.right$
return x

Минимум и максимум

Элемент с минимальным значением ключа всегда можно найти, следуя по дочерним указателям *left* от корневого узла до тех пор, пока не встретится значение NIL, как показано на рис. 12.2. Приведенная ниже процедура возвращает указатель на минимальный элемент поддерева с корнем в данном узле x, который предполагается не равным NIL.

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

Свойство бинарного дерева поиска гарантирует корректность процедуры TREE-MINIMUM. Если у узла x нет левого поддерева, то, поскольку все ключи в правом поддереве x не меньше ключа $x.key$, минимальный ключ поддерева с корнем в узле x находится в узле $x.key$. Если же у узла x есть левое поддерево, то, поскольку в правом поддереве не может быть узла с ключом, меньшим $x.key$, а все ключи в узлах левого поддерева не превышают $x.key$, узел с минимальным значением ключа в поддереве с корнем x находится в поддереве, корнем которого является узел $x.left$.

Псевдокод процедуры TREE-MAXIMUM симметричен.

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

Обе эти процедуры находят минимальный (максимальный) элемент дерева за время $O(h)$, где h — высота дерева, поскольку, как и в процедуре TREE-SEARCH, последовательность проверяемых узлов образует простой нисходящий путь от корня дерева.

Предшествующий и последующий элементы

Иногда для заданного узла в бинарном дереве поиска требуется определить, какой узел следует за ним в отсортированной последовательности, определяемой порядком центрированного обхода бинарного дерева, и какой узел предшествует данному. Если все ключи различны, последующим по отношению к узлу x является узел с наименьшим ключом, большим $x.key$. Структура бинарного дерева поиска позволяет найти этот узел, даже не выполняя сравнение ключей. Приведенная далее процедура возвращает узел, следующий за узлом x в бинарном дереве поиска (если таковой существует), и NIL, если x обладает наибольшим ключом в бинарном дереве.

```
TREE-SUCCESSOR(x)
1 if x.right ≠ NIL
2     return TREE-MINIMUM(x.right)
3 y = x.p
4 while y ≠ NIL и x == y.right
5     x = y
6     y = y.p
7 return y
```

Код процедуры TREE-SUCCESSOR разбивается на две части. Если правое поддерево узла x непустое, то следующий за x элемент является крайним слева узлом в правом поддереве x, который выявляется в строке 2 вызовом процедуры TREE-MINIMUM($x.right$). Например, на рис. 12.2 следующим за узлом с ключом 15 является узел с ключом 17.

С другой стороны, как требуется показать в упр. 12.2.6, если правое поддерево узла x пустое и у x имеется следующий за ним элемент y, то y является наименьшим предком x, левый наследник которого также является предком x. На рис. 12.2 следующим за узлом с ключом 13 является узел с ключом 15. Для того чтобы найти y, мы просто поднимаемся вверх по дереву от x до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя. Это действие выполняется в строках 3–7 процедуры TREE-SUCCESSOR.

Время работы алгоритма TREE-SUCCESSOR в дереве высотой h составляет $O(h)$, поскольку мы либо движемся по простому пути вниз от исходного узла, либо по простому пути вверх. Процедура поиска предшествующего узла в дереве TREE-PREDECESSOR симметрична процедуре TREE-SUCCESSOR и также имеет время работы $O(h)$.

Даже если в дереве имеются узлы с одинаковыми ключами, мы можем просто определить последующий и предшествующий узлы как такие, которые возвращаются процедурами TREE-SUCCESSOR(x) и TREE-PREDECESSOR(x) соответственно.

Таким образом, мы доказали следующую теорему.

Теорема 12.2

Операции SEARCH, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR над динамическим множеством могут быть реализованы таким образом, что их время выполнения равно $O(h)$ в бинарном дереве поиска высотой h. ■

Упражнения

12.2.1

Предположим, что имеется ряд чисел от 1 до 1000, организованных в виде бинарного дерева поиска, и мы выполняем поиск числа 363. Какая из следующих последовательностей *не* может быть последовательностью проверяемых узлов?

а. 2, 252, 401, 398, 330, 344, 397, 363.

б. 924, 220, 911, 244, 898, 258, 362, 363.

в. 925, 202, 911, 240, 912, 245, 363.

г. 2, 399, 387, 219, 266, 382, 381, 278, 363.

д. 935, 278, 347, 621, 299, 392, 358, 363.

12.2.2

Разработайте рекурсивные версии процедур TREE-MINIMUM и TREE-MAXIMUM.

12.2.3

Разработайте процедуру TREE-PREDECESSOR.

12.2.4

Разбираясь с бинарными деревьями поиска, студент решил, что обнаружил их новое замечательное свойство. Предположим, что поиск ключа k в бинарном дереве поиска завершается в листе. Рассмотрим три множества: множество ключей слева от пути поиска A, множество ключей на пути поиска B и множество ключей справа от пути поиска C. Студент считает, что любые три ключа $a \in A$, $b \in B$ и $c \in C$ должны удовлетворять неравенству $a \leq b \leq c$. Приведите наименьший возможный контрпример, опровергающий предположение студента.

12.2.5

Покажите, что если узел в бинарном дереве поиска имеет два дочерних узла, то последующий за ним узел не имеет левого дочернего узла, а предшествующий ему — правого.

12.2.6

Рассмотрим бинарное дерево поиска T, все ключи которого различны. Покажите, что если правое поддерево узла x в бинарном дереве поиска T пустое и у x есть следующий за ним элемент y, то y является самым нижним предком x, левый дочерний узел которого также является предком x. (Вспомните, что каждый узел является собственным предком.)

12.2.7

Альтернативный центрированный обход бинарного дерева поиска с n узлами можно осуществить путем поиска минимального элемента дерева с помощью процедуры TREE-MINIMUM с последующим $n - 1$ вызовом процедуры TREE-SUCCESSOR. Докажите, что время работы такого алгоритма равно $\Theta(n)$.

12.2.8

Докажите, что какой бы узел ни был взят в качестве исходного в бинарном дереве поиска высотой h, для k последовательных вызовов процедуры TREE-SUCCESSOR потребуется время $O(k + h)$.

12.2.9

Пусть T представляет собой бинарное дерево поиска с различными ключами, x — лист этого дерева, а y — его родительский узел. Покажите, что $y.key$ либо является наименьшим ключом в дереве T, превышающим ключ $x.key$, либо наибольшим ключом в T, меньшим ключа $x.key$.

12.3. Вставка и удаление

Операции вставки и удаления приводят к внесению изменений в динамическое множество, представленное бинарным деревом поиска. Структура данных должна быть изменена таким образом, чтобы отражать эти изменения, но при этом сохранить свойство бинарных деревьев поиска. Как мы увидим в этом разделе, вставка нового элемента в бинарное дерево поиска выполняется относительно просто, однако с удалением придется повозиться.

Вставка

Для вставки нового значения v в бинарное дерево поиска T мы воспользуемся процедурой TREE-INSERT. Процедура получает в качестве параметра узел z, атрибуты которого $z.key = v$, $z.left$ = NIL и $z.right$ = NIL. Процедура таким образом изменяет T и некоторые поля z, что z оказывается вставленным в корректную позицию дерева.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // Дерево T было пустым
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

На рис. 12.3 показано, как работает процедура TREE-INSERT. Подобно процедурам TREE-SEARCH и ITERATIVE-TREE-SEARCH, процедура TREE-INSERT начинает работу с корневого узла дерева и проходит по простому нисходящему пути. Указатель x отмечает путь, проходимый в поисках NIL, который должен быть заменен входным элементом z. Процедура поддерживает также ***замыкающий указатель*** (trailing pointer) y, который представляет собой указатель на родительский по отношению к x узел. После инициализации цикл **while** в строках 3–7 перемещает эти указатели вниз по дереву, перемещаясь влево или вправо в зависимости от результата сравнения ключей $z.key$ и $x.key$, до тех пор, пока x не станет равным NIL. Это значение находится именно в той позиции, в которую следует поместить элемент z. Замыкающий указатель y нужен для того, чтобы знать, какой узел должен быть изменен. В строках 8–13 выполняется установка значений указателей, обеспечивающая вставку элемента z.

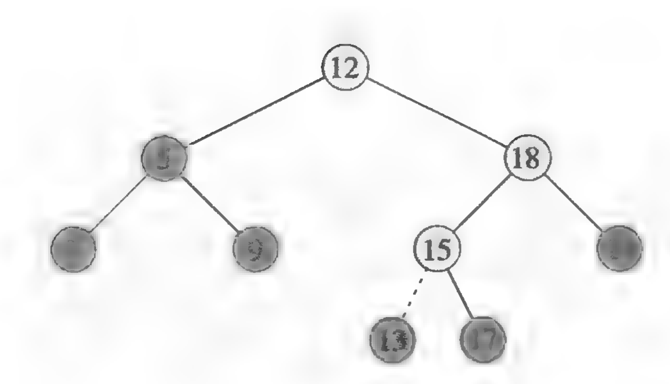


Рис. 12.3. Вставка элемента с ключом 13 в бинарное дерево поиска. Светлые узлы указывают простой путь от корня вниз по дереву в позицию, в которую должен быть вставлен новый элемент. Пунктирная линия указывает добавляемую в дерево связь, обеспечивающую вставку элемента.

Так же, как и другие примитивные операции над бинарным деревом поиска, процедура TREE-INSERT выполняется за время $O(h)$ в дереве высотой h.

Удаление

Стратегия удаления узла z из бинарного дерева поиска T имеет три основные ситуации; как мы увидим, одна из ситуаций оказывается достаточно сложной.

- Если у z нет дочерних узлов, то мы просто удаляем его, внося изменения в его родительский узел, а именно — заменяя дочерний узел z на NIL.
- Если у z только один дочерний узел, то мы удаляем узел z, создавая новую связь между родительским и дочерним узлами узла z.
- Если у узла z два дочерних узла, то мы находим следующий за ним узел y, который должен находиться в правом поддереве z и который занимает в дереве место z. Остаток исходного правого поддерева z становится новым поддеревом y, а левое поддерево z становится новым левым поддеревом y. Это самый сложный случай, поскольку, как мы увидим, здесь играет роль, является ли y правым дочерним узлом z.

Процедура удаления данного узла z из бинарного дерева поиска T получает в качестве аргумента указатели на T и на z. Она организована несколько иначе, чем описано ранее, и рассматривает не три, а четыре случая, показанные на рис. 12.4.

- Если z не имеет левого дочернего узла (см. рис. 12.4, (а)), то мы заменяем z его правым дочерним узлом, который может быть (или не быть) NIL. Если правый дочерний узел z представляет собой NIL, то мы оказываемся в ситуации, когда у узла z нет дочерних узлов. Если правый дочерний узел z отличен от NIL, то мы оказываемся в ситуации, когда у узла z имеется единственный дочерний узел, являющийся правым дочерним узлом.
- Если z имеет только один дочерний узел, являющийся его левым дочерним узлом (см. рис. 12.4, (б)), то мы заменяем z его левым дочерним узлом.
- В противном случае z имеет и левый, и правый дочерние узлы. Мы находим узел y, следующий за z. Этот узел располагается в правом поддереве z и не

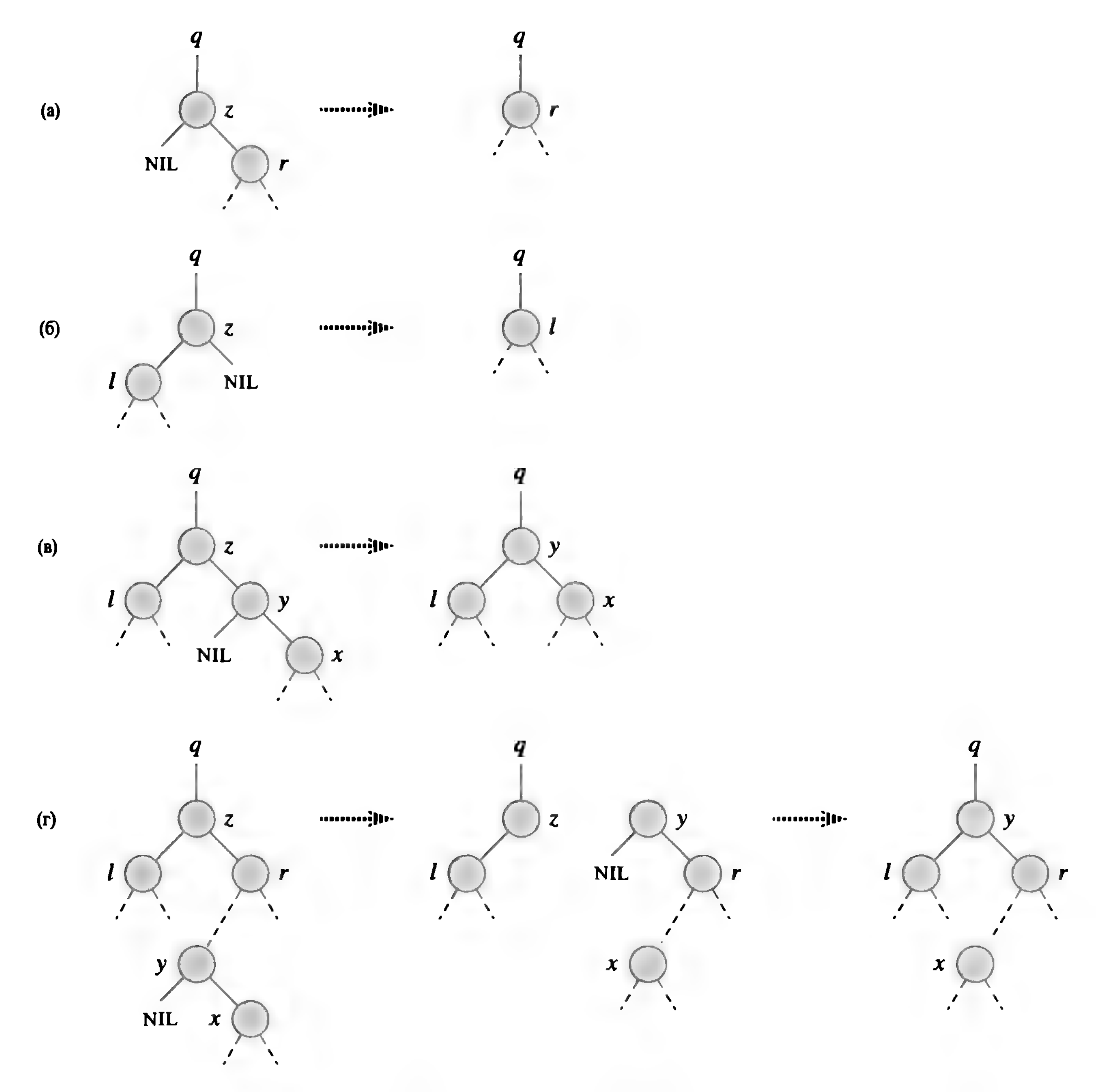


Рис. 12.4. Удаление узла z из бинарного дерева поиска. Узел z может быть корнем, левым дочерним узлом узла q или правым дочерним узлом q. **(а)** Узел z не имеет левого дочернего узла. Мы заменяем z его правым дочерним узлом r, который может быть (а может и не быть) значением NIL. **(б)** Узел z имеет левый дочерний узел l, но не имеет правого дочернего узла. Мы заменяем z на l. **(в)** Узел z имеет два дочерних узла; его левый дочерний узел — l, а правый дочерний узел y является следующим за z узлом дерева; правый дочерний по отношению к y узел — x. Мы заменяем z на y, обновляем левый дочерний узел y (им становится l), но оставляем x правым дочерним узлом y. **(г)** Узел z имеет два дочерних узла (левый дочерний узел l и правый дочерний узел r), а следующий за z узел $y \neq r$ находится в поддереве, корнем которого является r. Мы заменяем y его собственным правым дочерним узлом x и устанавливаем y в качестве родительского по отношению к r узла. Затем мы устанавливаем y в качестве дочернего узла q и родительского узла l.

имеет левого дочернего узла (см. упр. 12.2.5). Мы хотим вырезать y из его текущего положения и заменить им в дереве узел z.

- Если y является правым дочерним узлом z (см. рис. 12.4, (в)), то мы заменяем z на y, оставляя нетронутым правый дочерний по отношению к y узел.
- В противном случае y находится в правом поддереве узла z, но не является правым дочерним узлом z (см. рис. 12.4, (г)). В этом случае мы сначала заменяем y его собственным правым дочерним узлом, а затем заменяем z на y.

Для перемещения поддеревьев в бинарном дереве поиска мы определяем подпрограмму TRANSPLANT, которая заменяет одно поддерево, являющееся дочерним по отношению к своему родителю, другим поддеревом. Когда TRANSPLANT заменяет поддерево с корнем в узле u поддеревом с корнем в узле v, родитель узла u становится родителем узла v, который становится соответствующим дочерним узлом родительского по отношению к u узла.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

В строках 1 и 2 обрабатывается случай, когда u является корнем T. В противном случае u является либо левым, либо правым дочерним узлом своего родителя. В строках 3 и 4 выполняется обновление $u.p.left$, если u является левым дочерним узлом, в строке 5 обновляется $u.p.right$, если u является правым дочерним узлом. Мы допускаем значение NIL для узла v, и строки 6 и 7 обновляют $v.p$, если v не NIL. Заметим, что TRANSPLANT не пытается обновлять $v.left$ и $v.right$; выполнение этих действий (или их не выполнение) находится в зоне ответственности вызывающей подпрограмму TRANSPLANT процедуры.

Имея подпрограмму TRANSPLANT, мы можем реализовать процедуру удаления узла z из бинарного дерева поиска T следующим образом.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Процедура TREE-DELETE работает следующим образом. Строки 1 и 2 обрабатывают случай, когда у узла z нет левого дочернего узла, а строки 3 и 4 — когда у z есть левый дочерний узел, но нет правого. Строки 5–12 работают с оставшимися двумя случаями, когда у z есть два дочерних узла. Строка 5 находит узел y, следующий за z. Поскольку z имеет непустое правое поддерево, следующий за z узел должен быть узлом этого поддерева с наименьшим ключом; поэтому поиск узла y осуществляется вызовом TREE-MINIMUM($z.right$). Как отмечалось ранее, у y нет левого дочернего узла. Мы хотим вырезать y из его текущего положения, после чего этот узел должен заменить в дереве узел z. Если y является правым дочерним узлом z, то строки 10–12 заменяют z как дочерний узел его родителя на y и заменяют левый дочерний узел y левым дочерним узлом z. Если y не является правым дочерним узлом z, в строках 7–9 выполняется замена y как дочернего узла своего родителя правым дочерним по отношению к y узлом, и преобразование правого дочернего узла z в правый дочерний узел y, после чего строки 10–12 заменяют z как дочерний узел его родителя узлом y, а левым дочерним узлом y становится левый дочерний узел z.

Каждая строка TREE-DELETE, включая вызовы TRANSPLANT, выполняется за константное время, за исключением вызова TREE-MINIMUM в строке 5. Таким образом, TREE-DELETE выполняется за время $O(h)$ в дереве высотой h.

Таким образом, доказана следующая теорема.

Теорема 12.3

Операции над динамическим множеством INSERT и DELETE можно реализовать таким образом, что каждая из них выполняется в бинарном дереве поиска высотой h за время $O(h)$. ■

Упражнения

12.3.1

Приведите рекурсивную версию процедуры TREE-INSERT.

12.3.2

Предположим, что бинарное дерево поиска строится путем многократной вставки в дерево различных значений. Покажите, что количество узлов, проверяемых при поиске некоторого значения в дереве, на один больше, чем количество узлов, проверявшихся при вставке этого значения в дерево.

12.3.3

Отсортировать заданное множество из n чисел можно следующим образом: сначала построить бинарное дерево поиска, содержащее эти числа (многократно вызывая процедуру TREE-INSERT для вставки чисел в дерево одно за другим), а затем выполнить центрированный обход полученного дерева. Чему равно время работы такого алгоритма в наилучшем и наихудшем случаях?

12.3.4

Является ли операция удаления "коммутативной" в том смысле, что удаление x с последующим удалением y из бинарного дерева поиска приводит к тому же результирующему дереву, что и удаление y с последующим удалением x? Поясните, почему это так, или приведите контрпример.

12.3.5

Предположим, что вместо поддержки в каждом узле x атрибута $x.p$, указывающего на родительский узел x, поддерживается атрибут $x.succ$, указывающий на узел, следующий за x. Запишите псевдокод процедур SEARCH, INSERT и DELETE для бинарного дерева поиска, использующего такое представление. Эти процедуры должны выполняться за время $O(h)$, где h — высота дерева T. (*Указание:* можно реализовать подпрограмму для поиска узла, родительского по отношению к данному.)

12.3.6

Если узел z в процедуре TREE-DELETE имеет два дочерних узла, можно выбрать не следующий за ним узел y, а предшествующий ему. Какие при этом следует внести изменения в процедуру TREE-DELETE? Есть также мнение, что стратегия, которая состоит в равновероятном выборе предшествующего или последующего узла, приводит к большей производительности. Каким образом следует изменить процедуру TREE-DELETE для реализации указанной стратегии?

★ 12.4. Случайное построение бинарных деревьев поиска

Мы показали, что все базовые операции с бинарными деревьями поиска имеют время выполнения $O(h)$, где h — высота дерева. Однако при вставке и удалении элементов высота дерева меняется. Если, например, все элементы вставляются в дерево в строго возрастающей последовательности, то такое дерево вырождается в цепочку высотой $n - 1$. С другой стороны, как показано в упр. Б.5.4, $h \geq \lfloor \lg n \rfloor$. Как и в случае быстрой сортировки, можно показать, что поведение

алгоритма в среднем случае гораздо ближе к наилучшему случаю, чем к наихудшему.

К сожалению, в ситуации, когда при формировании бинарного дерева поиска используются и вставки, и удаления, о средней высоте образующихся деревьев известно мало, так что мы ограничимся анализом ситуации, когда дерево строится только с использованием вставок, без удалений. Определим ***случайно построенное бинарное дерево поиска*** (randomly built binary search tree) с n ключами как дерево, которое возникает при вставке ключей в изначально пустое дерево в случайном порядке, когда все $n!$ перестановок входных ключей равновероятны (в упр. 12.4.3 требуется показать, что это условие отличается от условия равновероятности всех возможных бинарных деревьев поиска с n узлами). В данном разделе мы докажем следующую теорему.

Теорема 12.4

Математическое ожидание высоты случайно построенного бинарного дерева поиска с n различными ключами равно $O(\lg n)$.

Доказательство. Начнем с определения трех случайных величин, которые помогут определить высоту случайного бинарного дерева поиска. Обозначая высоту случайного бинарного дерева поиска с n ключами как X_n, определим ***экспоненциальную высоту*** $Y_n = 2^{X_n}$. При построении бинарного дерева поиска с n ключами мы выбираем один из них в качестве корня. Обозначим через R_n случайную величину, равную ***рангу*** корневого ключа в множестве из всех n ключей, т.е. R_n содержит позицию, которую бы занимал ключ, если бы множество было отсортировано. Значение R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$. Если $R_n = i$, то левое поддерево корня представляет собой случайно построенное бинарное дерево поиска с $i - 1$ ключами, а правое — с $n-i$ ключами. Поскольку высота бинарного дерева на единицу больше наибольшей из высот поддеревьев корневого узла, экспоненциальная высота бинарного дерева в два раза больше экспоненциальной высоты наивысшего из поддеревьев корневого узла. Если мы знаем, что $R_n = i$, то

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) \ .$$

В качестве базового случая мы имеем $Y_1 = 1$, поскольку экспоненциальная высота дерева с одним узлом составляет $2^0 = 1$, и для удобства мы определим $Y_0 = 0$.

Далее мы определим индикаторные случайные величины $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, где

$$Z_{n,i} = \mathrm{I}\{R_n = i\} \ .$$

Поскольку R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$, $\Pr\{R_n = i\} = 1/n$ для $i = 1, 2, \ldots, n$, а следовательно, согласно лемме 5.1 мы имеем

$$\mathrm{E}[Z_{n,i}] = 1/n \qquad (12.1)$$

для $i = 1, 2, \ldots, n$. Поскольку ровно одно значение $Z_{n,i}$ равно 1, а все прочие равны 0, мы также имеем

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right) .$$

Мы покажем, что $\mathrm{E}[Y_n]$ полиномиально зависит от n, что неизбежно приводит к выводу, что $\mathrm{E}[X_n] = O(\lg n)$.

Мы утверждаем, что индикаторная случайная переменная $Z_{n,i} = \mathrm{I}\{R_n = i\}$ не зависит от значений Y_{i-1} и Y_{n-i}. Если $R_n = i$, то левое поддерево, экспоненциальная высота которого равна Y_{i-1}, случайным образом строится из $i-1$ ключей, ранги которых меньше i. Это поддерево ничем не отличается от любого другого случайного бинарного дерева поиска из $i-1$ ключей. Выбор $R_n = i$ никак не влияет на структуру этого дерева, а влияет только на количество содержащихся в нем узлов. Следовательно, случайные величины Y_{i-1} и $Z_{n,i}$ независимы. Аналогично правое поддерево, экспоненциальная высота которого равна Y_{n-i}, строится случайным образом из $n-i$ ключей, ранги которых больше i. Структура этого дерева не зависит от R_n, так что случайные величины Y_{n-i} и $Z_{n,i}$ независимы. Следовательно, мы имеем

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] && \text{(из линейности математического ожидания)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\, \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(из независимости)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(согласно (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1}, Y_{n-i})] && \text{(согласно (В.22))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(из упр. В.3.4)} .
\end{aligned}$$

Поскольку каждый член $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ появляется в последней сумме дважды, как $\mathrm{E}[Y_{i-1}]$ и как $\mathrm{E}[Y_{n-i}]$, мы получаем следующее рекуррентное соотношение:

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] . \tag{12.2}$$

Используя метод подстановок, покажем, что для всех натуральных n рекуррентное соотношение (12.2) имеет следующее решение:

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}.$$

При этом мы воспользуемся тождеством

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}. \tag{12.3}$$

(В упр. 12.4.1 данное тождество предлагается доказать самостоятельно.)

Заметим, что для базовых случаев границы $0 = Y_0 = \mathrm{E}[Y_0] \le (1/4)\binom{3}{3} = 1/4$ и $1 = Y_1 = \mathrm{E}[Y_1] \le (1/4)\binom{1+3}{3} = 1$ справедливы. По индукции имеем

$$\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&\le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(согласно гипотезе индукции)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(согласно (12.3))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned}$$

Мы получили границу для $\mathrm{E}[Y_n]$, но наша конечная цель — найти границу $\mathrm{E}[X_n]$. В упр. 12.4.4 требуется показать, что функция $f(x) = 2^x$ выпуклая вниз (см. с. 1251). Таким образом, мы можем применить неравенство Йенсена (В.26), которое гласит, что

$$\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \mathrm{E}\left[2^{X_n}\right] \\
&= \mathrm{E}[Y_n],
\end{aligned}$$

и получить

$$\begin{aligned} 2^{\mathrm{E}[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\ &= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3+6n^2+11n+6}{24}\,. \end{aligned}$$

Взятие логарифма от обеих частей дает нам $\mathrm{E}[X_n] = O(\lg n)$. ■

Упражнения

12.4.1

Докажите уравнение (12.3).

12.4.2

Приведите пример бинарного дерева поиска с n узлами, такого, что средняя глубина узла в дереве равна $\Theta(\lg n)$, в то время как высота дерева — $\omega(\lg n)$. Найдите асимптотическую верхнюю границу высоты бинарного дерева поиска с n узлами, средняя глубина узла в котором составляет $\Theta(\lg n)$.

12.4.3

Покажите, что понятие случайного бинарного дерева поиска с n ключами, когда выбор каждого дерева равновероятен, отличается от понятия случайно построенного бинарного дерева поиска, приведенного в этом разделе. (*Указание:* рассмотрите все возможные деревья при $n = 3$.)

12.4.4

Покажите, что функция $f(x) = 2^x$ является выпуклой вниз.

12.4.5 ★

Рассмотрим процедуру RANDOMIZED-QUICKSORT, работающую с входной последовательностью из n различных чисел. Докажите, что для любой константы $k > 0$ время работы алгоритма превышает $O(n \lg n)$ только для $O(1/n^k)$-й части всех возможных $n!$ перестановок.

Задачи

12.1. Бинарные деревья поиска с одинаковыми ключами

Одинаковые ключи приводят к проблеме при реализации бинарных деревьев поиска.

а. Чему равно асимптотическое время работы процедуры TREE-INSERT при вставке n одинаковых ключей в изначально пустое дерево?

Мы предлагаем улучшить алгоритм TREE-INSERT, добавив в него перед строкой 5 проверку равенства $z.key = x.key$, а перед строкой 11 — проверку равенства $z.key = y.key$. Если равенства выполняются, мы реализуем одну из описанных далее стратегий. Для каждой из них найдите асимптотическое время работы при вставке n одинаковых ключей в изначально пустое дерево. (Описания приведены для строки 5, в которой сравниваются ключи z и x. Для строки 11 замените в описании x на y.)

б. Храним в узле x булев флаг $x.b$ и устанавливаем x равным либо $x.left$, либо $x.right$, в зависимости от значения $x.b$, которое поочередно принимает значения FALSE и TRUE при каждом посещении x в процессе вставки узла с тем же ключом, что и у x.

в. Храним все элементы с одинаковыми ключами в одном узле x с помощью списка и при вставке просто добавляем элемент z в этот список.

г. Случайным образом присваиваем x значение $x.left$ или $x.right$. (Каково будет время работы такой стратегии в наихудшем случае? Оцените ожидаемое время работы данной стратегии.)

12.2. Цифровые деревья

Пусть имеются две строки $a = a_0a_1 \ldots a_p$ и $b = b_0b_1 \ldots b_q$, в которых все символы a_i и b_j принадлежат некоторому упорядоченному множеству. Мы говорим, что строка a ***лексикографически меньше*** строки b, если выполняется одно из двух условий:

1. существует целое $0 \leq j \leq \min(p, q)$, такое, что $a_i = b_i$ для всех $i = 0, 1, \ldots, j-1$ и $a_j < b_j$, или
2. $p < q$ и $a_i = b_i$ для всех $i = 0, 1, \ldots, p$.

Например, если a и b представляют собой битовые строки, то $10100 < 10110$ согласно правилу 1 (полагая $j = 3$), а $10100 < 101000$ согласно правилу 2. Это упорядочение подобно упорядочению слов в словаре естественного языка по алфавиту.

Структура данных ***цифрового дерева*** (radix tree), показанная на рис. 12.5, хранит битовые строки 1011, 10, 011, 100 и 0. При поиске ключа на глубине i мы переходим к левому узлу, если $a_i = 0$, и к правому, если $a_i = 1$. Пусть S — множество различных битовых строк с суммарной длиной n. Покажите, как использовать цифровое дерево для лексикографической сортировки за время $\Theta(n)$. В примере, приведенном на рис. 12.5, отсортированная последовательность должна выглядеть следующим образом: 0, 011, 10, 100, 1011.

12.3. Средняя глубина вершины в случайно построенном бинарном дереве поиска

В данной задаче мы докажем, что средняя глубина узла в случайно построенном бинарном дереве поиска с n узлами равна $O(\lg n)$. Хотя этот результат и является более слабым, чем в теореме 12.4, способ доказательства демонстрирует

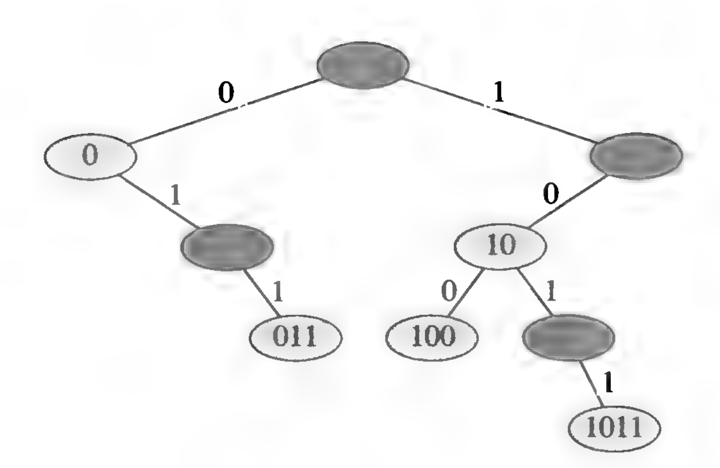


Рис. 12.5. Цифровое дерево с битовыми строками 1011, 10, 011, 100 и 0. Ключ каждого узла можно определить путем прохода по простому пути от корня к этому узлу. Следовательно, нет необходимости хранить ключи в узлах; здесь значения ключей приведены только в иллюстративных целях. Узлы заштрихованы темным цветом, если соответствующие им ключи отсутствуют в дереве; такие узлы существуют только для установления путей к другим узлам.

интересные аналогии между построением бинарного дерева поиска и работой алгоритма RANDOMIZED-QUICKSORT из раздела 7.3.

Определим ***общую длину путей*** $P(T)$ бинарного дерева T как сумму глубин всех узлов $x \in T$, которую мы будем обозначать как $d(x, T)$.

а. Покажите, что средняя глубина узла в дереве T равна

$$\frac{1}{n}\sum_{x \in T} d(x,T) = \frac{1}{n}P(T) .$$

Таким образом, нужно показать, что математическое ожидание $P(T)$ равно $O(n \lg n)$.

б. Обозначим через T_L и T_R соответственно левое и правое поддеревья дерева T. Покажите, что если дерево T имеет n узлов, то

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

в. Обозначим через $P(n)$ среднюю общую длину путей случайно построенного бинарного дерева поиска с n узлами. Покажите, что

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

г. Покажите, как переписать $P(n)$ как

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

д. Вспоминая альтернативный анализ рандомизированной версии алгоритма быстрой сортировки из задачи 7.3, сделайте вывод о том, что $P(n) = O(n \lg n)$.

В каждом рекурсивном вызове быстрой сортировки мы выбираем случайный опорный элемент для разделения множества сортируемых элементов. Каждый узел в бинарном дереве поиска также отделяет часть элементов, попадающих в поддерево, для которого данный узел является корневым.

е. Опишите реализацию алгоритма быстрой сортировки, в которой в процессе сортировки множества элементов выполняются те же сравнения, что и для вставки элементов в бинарное дерево поиска. (Порядок, в котором выполняются сравнения, может быть иным, однако сами множества сравнений должны совпадать.)

12.4. Количество различных бинарных деревьев

Обозначим через b_n количество различных бинарных деревьев с n узлами. В этой задаче будет выведена формула для b_n и найдена ее асимптотическая оценка.

а. Покажите, что $b_0 = 1$ и что для $n \geq 1$

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

б. Пусть $B(x)$ – производящая функция (определение производящей функции можно найти в задаче 4.4)

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Покажите, что $B(x) = xB(x)^2 + 1$, и, следовательно, $B(x)$ можно записать в аналитическом виде как

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

Разложение в ряд Тейлора функции $f(x)$ вблизи точки $x = a$ определяется как

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

где $f^{(k)}(x)$ – k-я производная f в точке x.

в. Покажите, что

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(n-е ***число Каталана***), используя разложение в ряд Тейлора функции $\sqrt{1-4x}$ вблизи точки $x = 0$. (Если хотите, вместо разложения в ряд Тейлора используйте обобщение биномиального разложения (В.4) для нецелого показателя степени n, когда для любого действительного числа n и целого k выражение $\binom{n}{k}$ интерпретируется как $n(n-1)\cdots(n-k+1)/k!$ при $k \geq 0$ и как 0 — в противном случае.)

г. Покажите, что
$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right) .$$

Заключительные замечания

Подробное рассмотрение простых бинарных деревьев поиска (которые были независимо открыты рядом исследователей в конце 1950-х годов) и их различных вариаций имеется в книге Кнута (Knuth) [210][1]. Там же рассматриваются и цифровые деревья. Цифровые деревья часто называют "лучами" ("tries"), термином, образованным из средних букв слова "по***луч***ение" ("re***trie***val").

Во многих книгах, включая два предыдущих издания данной книги, приводится несколько более простой метод удаления узла из бинарного дерева поиска при наличии обоих дочерних узлов. Вместо замены узла z следующим за ним узлом y мы удаляем узел y, но копируем его ключ и сопутствующие данные в узел z. Недостатком такого подхода является то, что в действительности удаляемый узел может не совпадать с узлом, передаваемым процедуре удаления. Если другие компоненты программы работают с указателями на узлы дерева, в результате они могут оказаться с устаревшими указателями на реально удаленные узлы. Хотя метод удаления, представленный в этом издании книги, немного сложнее использовавшегося ранее, он гарантирует, что вызов для удаления узла z в действительности удалит узел z и только его.

В разделе 15.5 будет показано, каким образом можно построить оптимальное бинарное дерево поиска, если частоты поисков известны заранее, до начала построения дерева. В этом случае дерево строится таким образом, чтобы при наиболее частых поисках просматривалось минимальное количество узлов дерева.

---

[1]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 13. Красно-черные деревья

В главе 12 было показано, что бинарные деревья поиска высоты h реализуют все базовые операции над динамическими множествами, такие как SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT и DELETE, со временем работы $O(h)$. Таким образом, операции выполняются тем быстрее, чем меньше высота дерева. Однако в наихудшем случае производительность бинарного дерева поиска оказывается ничуть не лучшей, чем производительность связанного списка. Красно-черные деревья представляют собой одну из множества "сбалансированных" схем деревьев поиска, которые гарантируют время выполнения операций над динамическим множеством $O(\lg n)$ даже в наихудшем случае.

13.1. Свойства красно-черных деревьев

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом ***цвета*** в каждом узле. Цвет узла может быть либо красным (RED), либо черным (BLACK). В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно ***сбалансированными***.

Каждый узел дерева содержит атрибуты *color*, *key*, *left*, *right* и *p*. Если не существует дочернего или родительского узла по отношению к данному, соответствующий указатель принимает значение NIL. Мы будем рассматривать эти значения NIL как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все "нормальные" узлы, содержащие поле ключа, становятся внутренними узлами дерева.

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим ***красно-черным свойствам***.

1. Каждый узел является либо красным, либо черным.
2. Корень дерева является черным узлом.
3. Каждый лист дерева (NIL) является черным узлом.
4. Если узел красный, то оба его дочерних узла черные.

5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

На рис. 13.1, (а) приведен пример красно-черного дерева.

Для удобства работы с граничными условиями в красно-черных деревьях мы заменим все листья одним ограничивающим узлом, представляющим значение NIL (с этим приемом мы уже встречались — см. с. 270). В красно-черном дереве T ограничитель $T.nil$ представляет собой объект с теми же атрибутами, что и обычный узел дерева. Значение *color* этого узла равно BLACK (черный), а все остальные атрибуты — p, *left*, *right* и *key* — могут иметь произвольные значения. Как показано на рис. 13.1, (б), все указатели на NIL заменяются указателями на ограничитель $T.nil$.

Использование ограничителя позволяет нам рассматривать дочерний по отношению к узлу x NIL как обычный узел, родителем которого является узел x. Хотя можно было бы использовать различные ограничители для каждого значения NIL (что позволило бы точно определять их родительские узлы), этот подход привел бы к неоправданному перерасходу памяти. Вместо этого мы используем единственный ограничитель $T.nil$ для представления всех NIL — как листьев, так и родительского узла корня. Значения атрибутов p, *left*, *right* и *key* ограничителя не играют никакой роли, хотя для удобства мы можем присвоить им те или иные значения.

В целом мы ограничим наш интерес к красно-черным деревьям только их внутренними узлами, поскольку лишь они хранят значения ключей. В оставшейся части данной главы при изображении красно-черных деревьев все листья опускаются, как это сделано на рис. 13.1, (в).

Количество черных узлов на любом простом пути от узла x (не считая сам узел) к листу будем называть ***черной высотой*** (black-height) узла и обозначать как $\text{bh}(x)$. В соответствии со свойством 5 красно-черных деревьев черная высота узла — точно определяемое значение, поскольку все нисходящие простые пути из узла содержат одно и то же количество черных узлов. Черной высотой дерева будем считать черную высоту его корня.

Следующая лемма показывает, почему красно-черные деревья хорошо использовать в качестве деревьев поиска.

Лемма 13.1

Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\lg(n+1)$.

Доказательство. Начнем с того, что покажем, что поддерево любого узла x содержит как минимум $2^{\text{bh}(x)} - 1$ внутренних узлов. Докажем это по индукции по высоте x. Если высота x равна 0, то узел x должен быть листом ($T.nil$), а поддерево узла x содержит не менее $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов. Теперь для выполнения шага индукции рассмотрим узел x, который имеет положительную высоту и представляет собой внутренний узел с двумя потомками. Каждый дочерний узел имеет черную высоту либо $\text{bh}(x)$, либо $\text{bh}(x) - 1$ в зависимости от того, является ли его цвет соответственно красным или черным. Поскольку

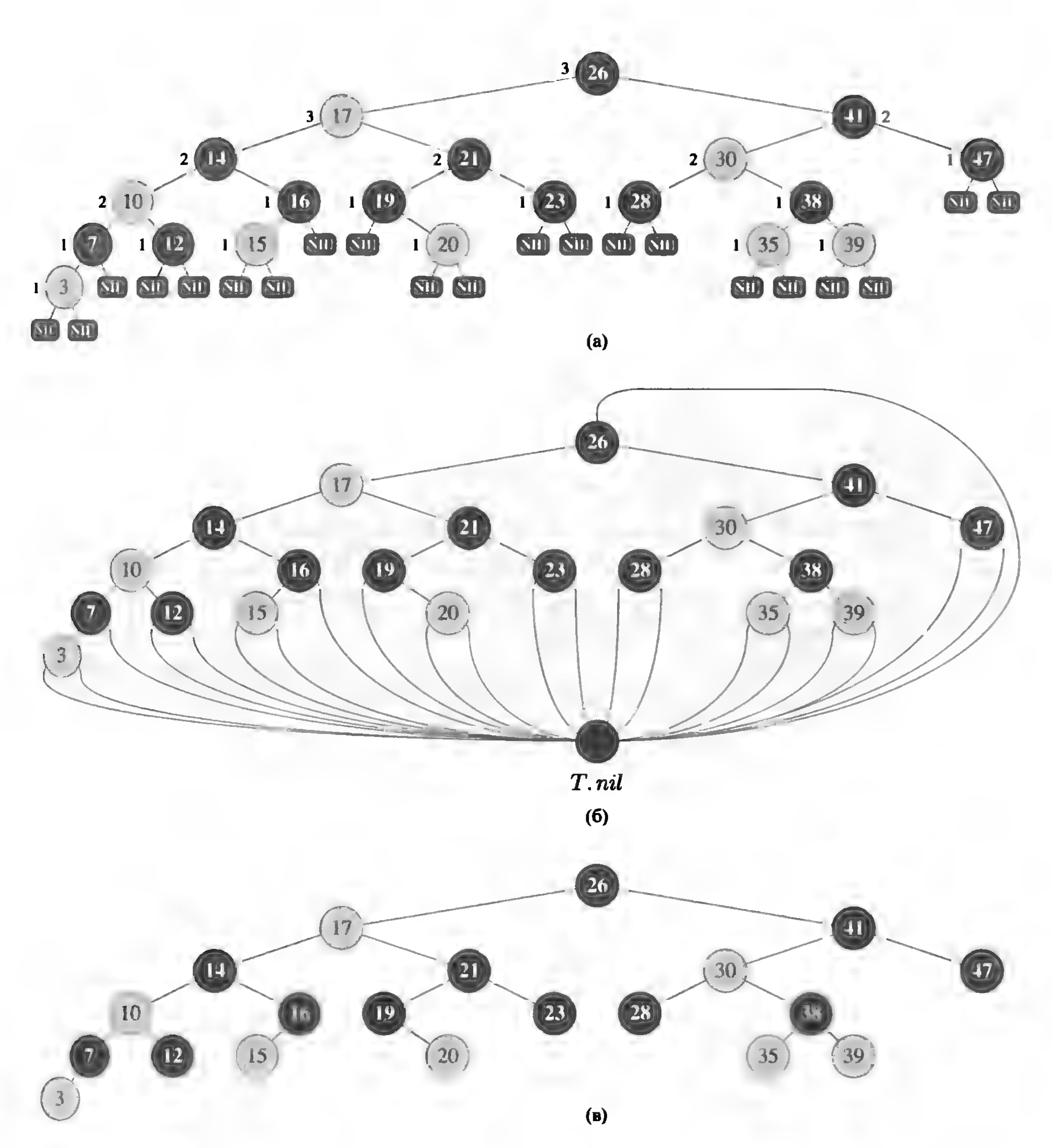


Рис. 13.1. Красно-черное дерево с более темными черными узлами и более светлыми красными. Каждый узел красно-черного дерева либо красный, либо черный; оба дочерних узла красного узла — черные, и количество черных узлов одинаково на каждом простом пути от узла до его наследника-листа. **(а)** Каждый лист, показанный как NIL, черный. Каждый не-NIL узел помечен его черной высотой; листья NIL имеют черную высоту 0. **(б)** То же красно-черное дерево, но все NIL в нем заменены единственным ограничителем $T.nil$, который всегда черный, а черные высоты узлов опущены. Родителем корневого узла также является ограничитель. **(в)** То же красно-черное дерево, но с опущенными листьями и родителем корневого узла. Именно этот стиль изображения красно-черных деревьев будет использоваться далее в этой главе.

высота потомка x меньше, чем высота самого узла x, мы можем использовать предположение индукции и сделать вывод о том, что каждый из потомков x имеет как минимум $2^{\text{bh}(x)-1} - 1$ внутренних узлов. Таким образом, дерево с корнем в вершине x содержит как минимум $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ внутренних узлов, что и доказывает наше утверждение.

Для того чтобы завершить доказательство леммы, обозначим высоту дерева через h. Согласно свойству 4 по крайней мере половина узлов на любом простом пути от корня к листу, не считая сам корень, должны быть черными. Следовательно, черная высота корня должна составлять как минимум $h/2$; значит,

$$n \geq 2^{h/2} - 1 .$$

Перенося 1 в левую часть и логарифмируя, получим, что $\lg(n + 1) \geq h/2$, или $h \leq 2\lg(n + 1)$. ■

Непосредственным следствием леммы является то, что такие операции над динамическими множествами, как SEARCH, MINIMUM, MAXIMUM, PREDECESSOR и SUCCESSOR, при использовании красно-черных деревьев выполняются за время $O(\lg h)$, поскольку, как показано в главе 12, время работы этих операций на дереве поиска высотой h составляет $O(h)$, а любое красно-черное дерево с n узлами является деревом поиска высотой $O(\lg n)$. (Само собой разумеется, обращения к NIL в алгоритмах в главе 12 должны быть заменены обращениями к $T.nil$.) Хотя алгоритмы TREE-INSERT и TREE-DELETE из главы 12 и характеризуются временем работы $O(\lg n)$, если использовать их для вставки и удаления из красно-черного дерева, непосредственно использовать их для выполнения операций INSERT и DELETE нельзя, поскольку они не гарантируют сохранение красно-черных свойств после внесения изменений в дерево. Однако в разделах 13.3 и 13.4 вы увидите, что эти операции также могут быть выполнены за время $O(\lg n)$.

Упражнения

13.1.1

Начертите в стиле рис. 13.1, (а) полное бинарное дерево поиска высотой 3 с ключами $\{1, 2, \ldots, 15\}$. Добавьте к нему листья NIL и раскрасьте полученное дерево разными способами, так, чтобы в результате получились красно-черные деревья с черной высотой 2, 3 и 4.

13.1.2

Начертите красно-черное дерево, которое получится в результате вставки с помощью алгоритма TREE-INSERT ключа 36 в дерево, изображенное на рис. 13.1. Если вставленный узел закрасить красным, будет ли полученное в результате дерево красно-черным? А если закрасить этот узел черным?

13.1.3

Определим ***ослабленное красно-черное дерево*** как бинарное дерево поиска, которое удовлетворяет красно-черным свойствам 1, 3, 4 и 5. Другими словами, корень

может быть как черным, так и красным. Рассмотрите ослабленное красно-черное дерево T, корень которого красный. Если мы перекрасим корень дерева T из красного цвета в черный, будет ли полученное в результате дерево красно-черным?

13.1.4

Предположим, что каждый красный узел в красно-черном дереве "поглощается" его черным родительским узлом, так что дочерний узел красного узла становится дочерним узлом его черного родителя (мы не обращаем внимания на то, что при этом происходит с ключами в узлах). Чему равен возможный порядок черного узла после того, как будут поглощены все его красные потомки? Что вы можете сказать о глубине листьев полученного дерева?

13.1.5

Покажите, что самый длинный простой нисходящий путь от вершины x к листу красно-черного дерева имеет длину, не более чем в два раза превышающую кратчайший нисходящий путь от x к листу-потомку.

13.1.6

Чему равно наибольшее возможное количество внутренних узлов в красно-черном дереве с черной высотой k? А наименьшее возможное количество?

13.1.7

Опишите красно-черное дерево с n ключами с наибольшим возможным отношением количества красных внутренних узлов к количеству черных внутренних узлов. Чему равно это отношение? Какое дерево имеет наименьшее указанное отношение, и чему равна его величина?

13.2. Повороты

Операции над деревом поиска TREE-INSERT и TREE-DELETE, будучи применены к красно-черному дереву с n ключами, выполняются за время $O(\lg n)$. Поскольку они изменяют дерево, в результате их работы могут нарушаться красно-черные свойства, перечисленные в разделе 13.1. Для восстановления этих свойств необходимо изменить цвета некоторых узлов дерева, а также структуру его указателей.

Изменения в структуре указателей будут выполняться с помощью ***поворотов*** (rotations), которые представляют собой локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска. На рис. 13.2 показаны два типа поворотов — левый и правый. При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом $T.nil$; x может быть любым узлом дерева, правый дочерний узел которого — не $T.nil$. Левый поворот выполняется "вокруг" связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

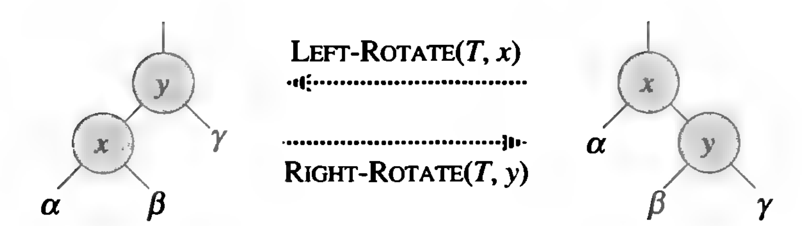


Рис. 13.2. Операция поворота в бинарном дереве поиска. Операция LEFT-ROTATE(T, x) преобразует конфигурацию из двух узлов справа в конфигурацию, показанную слева, путем изменения константного количества указателей. Обратная операция RIGHT-ROTATE(T, y) преобразует конфигурацию, показанную слева, в конфигурацию в правой части рисунка. Буквы α, β и γ представляют произвольные поддеревья. Операция поворота сохраняет свойство бинарного дерева поиска: ключи в α предшествуют ключу $x.key$, который предшествует ключам в β, которые предшествуют ключу $y.key$, который предшествует ключам в γ.

В псевдокоде процедуры LEFT-ROTATE предполагается, что $x.right \neq T.nil$ и что родитель корневого узла — $T.nil$.

```
LEFT-ROTATE(T, x)
y = x.right                 // Установка y
x.right = y.left            // Превращение левого поддерева y
                            // в правое поддерево x
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                   // Передача родителя x узлу y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                  // Размещение x в качестве левого
                            // дочернего узла y
x.p = y
```

На рис. 13.3 показан конкретный пример изменения бинарного дерева поиска процедурой LEFT-ROTATE. Код процедуры RIGHT-ROTATE симметричен коду LEFT-ROTATE. Обе эти процедуры выполняются за время $O(1)$. При повороте изменяются только указатели; все прочие атрибуты сохраняют свое значение.

Упражнения

13.2.1

Напишите псевдокод процедуры RIGHT-ROTATE.

13.2.2

Покажите, что в каждом бинарном дереве поиска с n узлами имеется ровно $n - 1$ возможных поворотов.

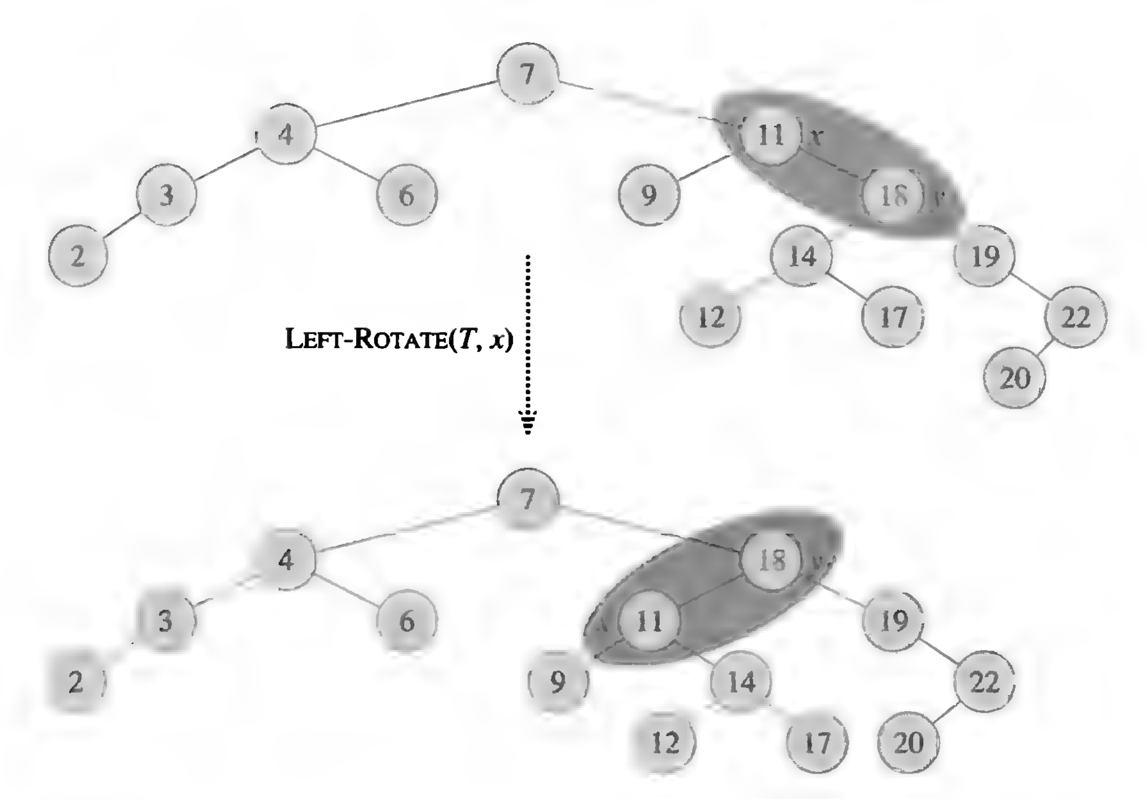


Рис. 13.3. Пример изменения бинарного дерева поиска процедурой LEFT-ROTATE(T, x). Центрированные обходы исходного и модифицированного деревьев дают одинаковые списки значений ключей.

13.2.3

Пусть a, b и c представляют собой произвольные узлы в поддеревьях α, β и γ соответственно в правом дереве на рис. 13.2. Как изменятся глубины узлов a, b и c при левом повороте в узле x?

13.2.4

Покажите, что произвольное бинарное дерево поиска с n узлами может быть преобразовано в любое другое бинарное дерево поиска с n узлами с использованием $O(n)$ поворотов. (*Указание:* сначала покажите, что $n - 1$ правых поворотов достаточно для преобразования дерева в правую цепочку.)

13.2.5 ★

Назовем бинарное дерево поиска T_1 ***правопреобразуемым*** в бинарное дерево поиска T_2, если можно получить T_2 из T_1 путем выполнения последовательности вызовов процедуры RIGHT-ROTATE. Приведите пример двух деревьев T_1 и T_2, таких, что T_1 не является правопреобразуемым в T_2. Затем покажите, что если дерево T_1 является правопреобразуемым в T_2, то это преобразование можно выполнить с помощью $O(n^2)$ вызовов процедуры RIGHT-ROTATE.

13.3. Вставка

Вставка узла в красно-черное дерево с n узлами может быть выполнена за время $O(\lg n)$. Для вставки узла z в дерево T мы используем немного модифицированную версию процедуры TREE-INSERT (см. раздел 12.3), которая вставляет узел в дерево, как если бы это было обычное бинарное дерево поиска, а затем окрашивает его в красный цвет. (В упр. 13.3.1 нужно ответить, почему выбран именно красный, а не черный цвет.) Для того чтобы вставка сохраняла красно-черные свойства дерева, после нее вызывается вспомогательная процедура RB-INSERT-FIXUP, которая перекрашивает узлы и выполняет повороты. Вызов RB-INSERT(T, z) вставляет в красно-черное дерево T узел z с уже заполненным атрибутом *key*.

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

Есть четыре отличия между процедурами TREE-INSERT и RB-INSERT. Во-первых, все NIL в TREE-INSERT заменены на $T.nil$. Во-вторых, для поддержки корректности структуры дерева в строках 14 и 15 процедуры RB-INSERT выполняется присвоение $T.nil$ атрибутам $z.left$ и $z.right$. В третьих, в строке 16 мы назначаем узлу z красный цвет. И наконец, в-четвертых, поскольку красный цвет z может вызвать нарушение одного из красно-черных свойств, в строке 17 вызывается вспомогательная процедура RB-INSERT-FIXUP(T, z), назначение которой — восстановить красно-черные свойства дерева.

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK                         // Случай 1
6              y.color = BLACK                           // Случай 1
7              z.p.p.color = RED                         // Случай 1
8              z = z.p.p                                 // Случай 1
9          else if z == z.p.right
10                 z = z.p                               // Случай 2
11                 LEFT-ROTATE(T, z)                     // Случай 2
12             z.p.color = BLACK                         // Случай 3
13             z.p.p.color = RED                         // Случай 3
14             RIGHT-ROTATE(T, z.p.p)                    // Случай 3
15     else (то же, что и в части then, но с заменой
               “правого” (right) “левым” (left) и наоборот)
16 T.root.color = BLACK
```

Для того чтобы понять, как работает процедура RB-INSERT-FIXUP, разобьем изучение кода на три части. Сначала определим, какие из красно-черных свойств нарушаются при вставке узла z и его окраске в красный цвет. Затем рассмотрим назначение цикла **while** в строках 1–15. После этого изучим каждый из трех случаев[1], которые встречаются в этом цикле, и посмотрим, каким образом достигается цель в каждом из них. На рис. 13.4 показан пример работы процедуры RB-INSERT-FIXUP.

Какие из красно-черных свойств могут быть нарушены перед вызовом RB-INSERT-FIXUP? Свойство 1, определенно, выполняется (как и свойство 3), так как оба дочерних узла вставляемого узла являются ограничителями $T.nil$. Свойство 5, согласно которому для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов, также остается в силе, поскольку узел z замещает (черный) ограничитель, будучи при этом красным и имея черные дочерние узлы. Таким образом, может нарушаться только свойство 2, которое требует, чтобы корень красно-черного дерева был черным, и свойство 4, согласно которому красный узел не может иметь красного потомка. Оба нарушения возможны в силу того, что узел z после вставки окрашивается в красный цвет. Свойство 2 оказывается нарушенным, если узел z становится корнем, а свойство 4 — если родительский по отношению к z узел является красным. На рис. 13.4, (а) показано нарушение свойства 4 после вставки узла z.

Цикл **while** в строках 1–15 сохраняет в начале каждой итерации цикла следующий инвариант, состоящий из трех частей.

---

[1] Случаи 2 и 3 не являются взаимоисключающими.

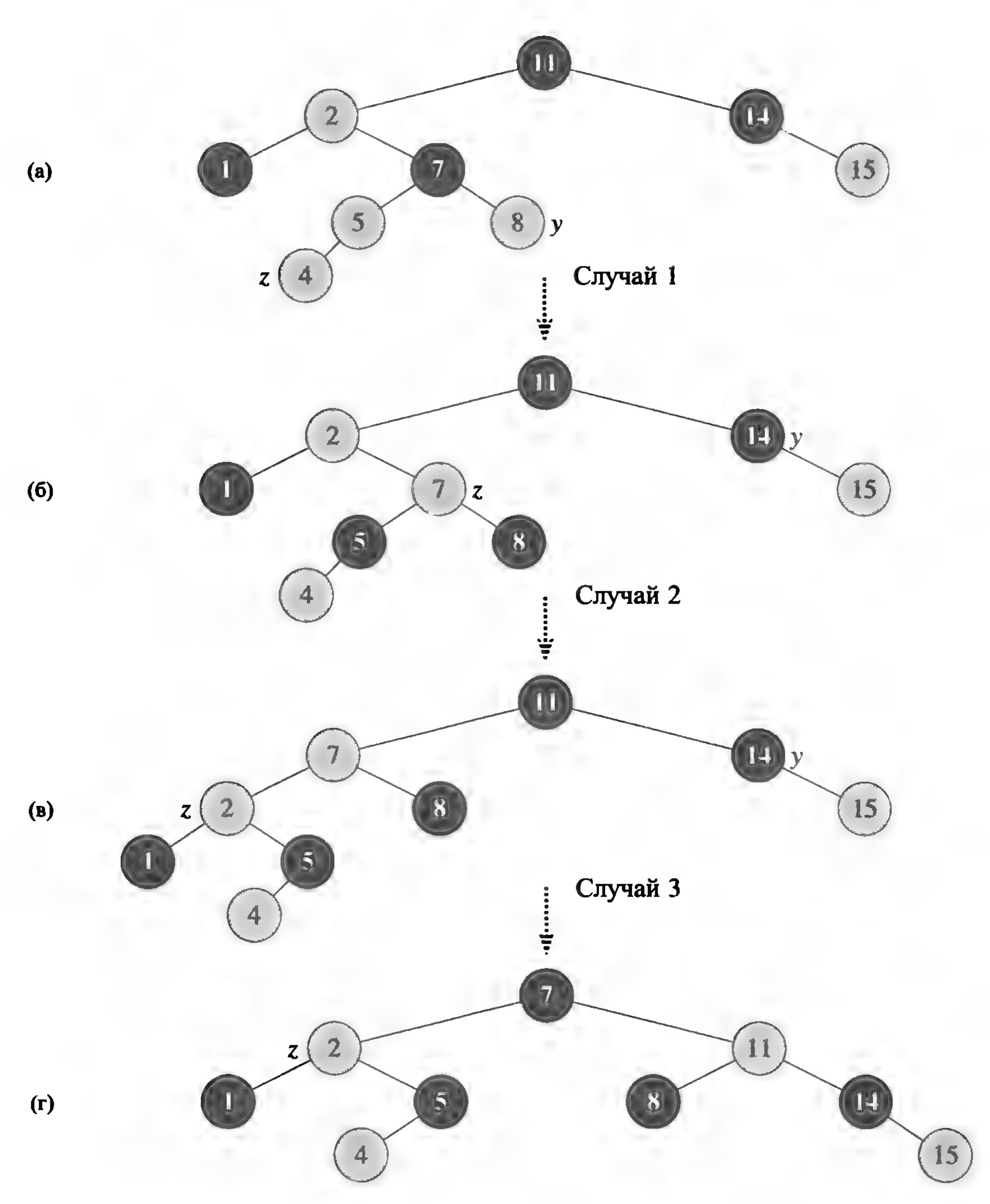


Рис. 13.4. Пример работы процедуры RB-INSERT-FIXUP. **(а)** Узел z после вставки. Поскольку и z, и его родитель $z.p$ красные, наблюдается нарушение свойства 4. Поскольку "дядя" z, узел y, красный, в коде процедуры срабатывает случай 1. Мы перекрашиваем узлы и перемещаем указатель z вверх по дереву, получая результат, показанный в части **(б)**. Вновь z и его родитель красные, но теперь "дядя" z, узел y, черный. Поскольку z является правым дочерним узлом по отношению к $z.p$, применим случай 2. Мы выполняем левый поворот, и полученное в результате дерево показано в части **(в)**. Теперь z является левым дочерним узлом своего родителя, так что применим случай 3. Перекрашивание и правый поворот дают показанное в части **(г)** корректное красно-черное дерево.

а. Узел z красный.

б. Если $z.p$ является корнем, то $z.p$ черный.

в. Если имеется нарушение красно-черных свойств, то это нарушение только одно — нарушение либо свойства 2, либо свойства 4. Если нарушено свойство 2, то это вызвано тем, что корнем дерева является красный узел z; если нарушено свойство 4, то в этом случае красными являются узлы z и $z.p$.

Часть (в), в которой говорится о возможных нарушениях красно-черных свойств, наиболее важна для того, чтобы показать, что процедура RB-INSERT-FIXUP восстанавливает красно-черные свойства. Части (а) и (б) просто поясняют ситуацию. Поскольку мы сосредоточиваем свое рассмотрение только на узле z и узлах, находящихся в дереве вблизи него, полезно знать, что узел z — красный (часть (а)). Часть (б) используется для того, чтобы показать, что узел $z.p.p$, к которому мы обращаемся в строках 2, 3, 7, 8, 13 и 14, существует.

Вспомним, что необходимо показать, что инвариант цикла выполняется перед первой итерацией цикла, что любая итерация цикла сохраняет инвариант и что инвариант цикла обеспечивает выполнение требуемого свойства по окончании работы цикла.

Начнем с рассмотрения инициализации и завершения работы цикла, а затем, подробнее рассмотрев работу цикла, докажем, что он сохраняет инвариант цикла. Попутно покажем, что есть только два возможных варианта действий в каждой итерации цикла — указатель z перемещается вверх по дереву или выполняются некоторые повороты и цикл завершается.

Инициализация. Перед выполнением первой итерации цикла имеется красно-черное дерево без каких-либо нарушений красно-черных свойств, к которому мы добавляем красный узел z. Покажем, что все части инварианта цикла выполняются к моменту вызова процедуры RB-INSERT-FIXUP.

а. В момент вызова процедуры RB-INSERT-FIXUP узел z — вставленный в дерево красный узел.

б. Если узел $z.p$ является корнем, то он черный и не изменяется до вызова процедуры RB-INSERT-FIXUP.

в. Мы уже убедились в том, что красно-черные свойства 1, 3 и 5 сохраняются к моменту вызова процедуры RB-INSERT-FIXUP.

Если нарушается свойство 2, то красный корень должен быть добавленным в дерево узлом z, который при этом является единственным внутренним узлом дерева. Поскольку и родитель, и оба потомка z являются ограничителями, свойство 4 не нарушается. Таким образом, нарушение свойства 2 — единственное нарушение красно-черных свойств во всем дереве.

Если же нарушено свойство 4, то, поскольку дочерние по отношению к z узлы являются черными ограничителями, а до вставки z никаких нарушений красно-черных свойств в дереве не было, нарушение заключается в том, что и z, и $z.p$ — красные. Кроме этого, других нарушений красно-черных свойств не имеется.

Завершение. Цикл завершает свою работу, когда $z.p$ становится черным (если z — корневой узел, то $z.p$ представляет собой черный ограничитель $T.nil$). Таким образом, свойство 4 при завершении цикла не нарушается. В соответствии с инвариантом цикла единственным нарушением красно-черных свойств может быть нарушение свойства 2. В строке 16 это свойство восстанавливается, так что по завершении работы процедуры RB-INSERT-FIXUP все красно-черные свойства дерева выполняются.

Сохранение. В действительности во время работы цикла **while** следует рассмотреть шесть разных случаев, однако три из них симметричны трем другим; разница лишь в том, является ли родитель $z.p$ левым или правым дочерним узлом по отношению к своему родителю $z.p.p$, что и выясняется в строке 2 (мы привели код только для ситуации, когда $z.p$ является левым потомком). Узел $z.p.p$ существует, поскольку, в соответствии с частью (б) инварианта цикла, если $z.p$ — корень дерева, то он черный. Поскольку цикл начинает работу, только если $z.p$ — красный, то $z.p$ не может быть корнем. Следовательно, $z.p.p$ существует.

Случай 1 отличается от случаев 2 и 3 цветом "брата" родительского по отношению к z узла, т.е. "дяди" узла z. После выполнения строки 3 указатель y указывает на дядю узла z — узел $z.p.p.right$, а в строке 4 проверяется его цвет. Если y — красный, выполняется код для случая 1; в противном случае выполняется код для случаев 2 и 3. В любом случае узел $z.p.p$ — черный, поскольку узел $z.p$ — красный, а свойство 4 нарушается только между z и $z.p$.

Случай 1. "Дядя" у узла z — красный

На рис. 13.5 показана ситуация, возникающая в случае 1 (строки 5–8), когда и $z.p$, и y — красные узлы. Поскольку узел $z.p.p$ — черный, мы можем исправить ситуацию, когда и z, и $z.p$ оба красные, покрасив и $z.p$, и y в черный цвет. Мы можем также окрасить $z.p.p$ в красный цвет, тем самым поддержав свойство 5. После этого мы повторяем цикл **while** с узлом $z.p.p$ в качестве нового узла z. Указатель z, таким образом, перемещается на два уровня вверх по дереву.

Теперь покажем, что в случае 1 инвариант цикла сохраняется. Обозначим через z узел z в текущей итерации, а через $z' = z.p.p$ — узел, который будет называться z в проверке в строке 1 в следующей итерации.

а. Поскольку в данной итерации цвет узла $z.p.p$ становится красным, в начале следующей итерации узел z' — красный.

б. Узел $z'.p$ в текущей итерации — $z.p.p.p$, и цвет данного узла в пределах данной итерации не изменяется. Если это корневой узел, то его цвет до начала данной итерации был черным и остается таковым в начале следующей итерации.

в. Мы уже доказали, что в случае 1 свойство 5 сохраняется; кроме того, понятно, что при выполнении итерации не возникает нарушения свойства 1 или 3.

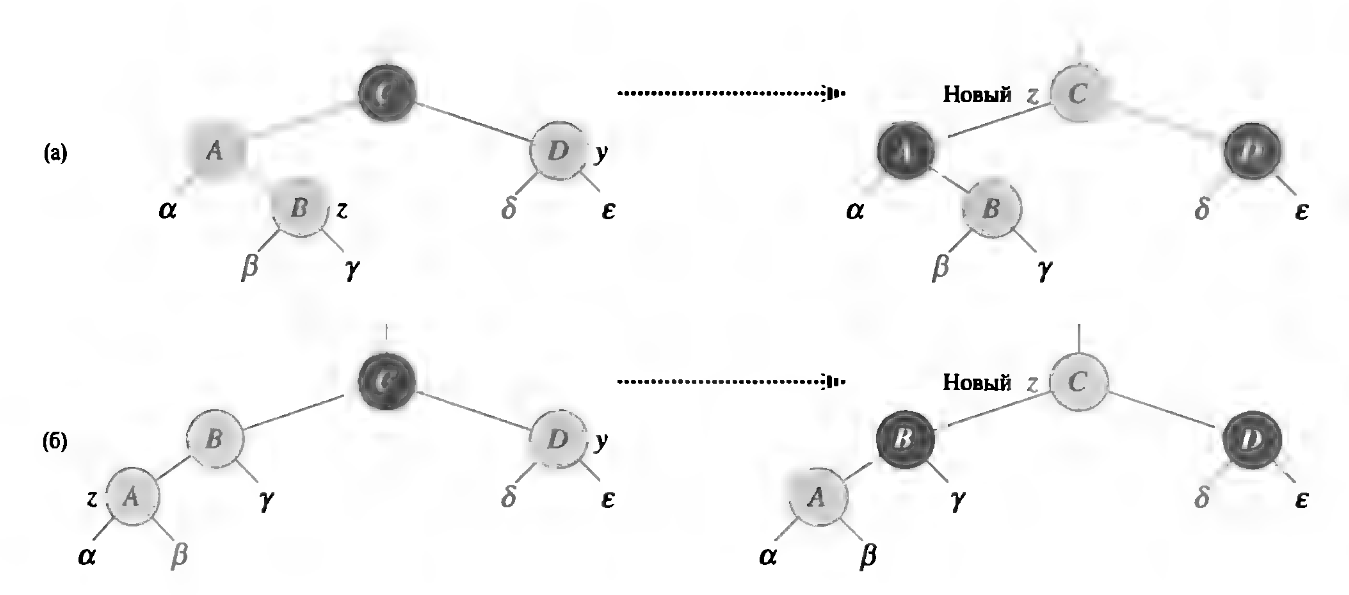


Рис. 13.5. Случай 1 процедуры RB-INSERT-FIXUP. Свойство 4 нарушено, поскольку и z, и его родительский узел $z.p$ красные. Мы предпринимаем одни и те же действия, когда **(а)** z является правым дочерним узлом и когда **(б)** z является левым дочерним узлом. Каждое из поддеревьев α, β, γ, δ и ε имеет черный корень, и у каждого одна и та же черная высота. Код для случая 1 изменяет цвета некоторых узлов, сохраняя свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Цикл **while** продолжается с "дедом" $z.p.p$ узла z в качестве нового узла z. Любое нарушение свойства 4 может теперь произойти только между новым z (окрашенным в красный цвет) и его родителем, если он также красный.

Если узел z' в начале очередной итерации является корнем, то код, соответствующий случаю 1, корректирует единственное нарушение свойства 4. Поскольку узел z' — красный и корневой, единственным нарушенным становится свойство 2, причем это нарушение связано с узлом z'.

Если узел z' в начале следующей итерации является корнем, то код, соответствующий случаю 1, корректирует единственное нарушение свойства 4, имеющееся перед выполнением итерации. Поскольку z' — узел красный и корневой, свойство 2 становится единственным нарушенным, и это нарушение вызвано узлом z'.

Если узел z' в начале следующей итерации корнем не является, то код, соответствующий случаю 1, не вызывает нарушения свойства 2. Этот код корректирует единственное нарушение свойства 4, имеющееся перед выполнением итерации. Коррекция выражается в том, что узел z' становится красным и оставляет узел $z'.p$ нетронутым. Если узел $z'.p$ был черным, то свойство 4 не нарушается; если же этот узел был красным, то окрашивание узла z' в красный цвет приводит к нарушению свойства 4 между узлами z' и $z'.p$.

Случай 2. "Дядя" у узла z черный, и z — правый потомок
Случай 3. "Дядя" у узла z черный, и z — левый потомок

В случаях 2 и 3 цвет узла y, являющегося "дядей" узла z, черный. Эти два случая отличаются один от другого тем, что z является левым или правым дочерним узлом по отношению к родительскому узлу $z.p$. Строки 10 и 11

12 Зак. 3726

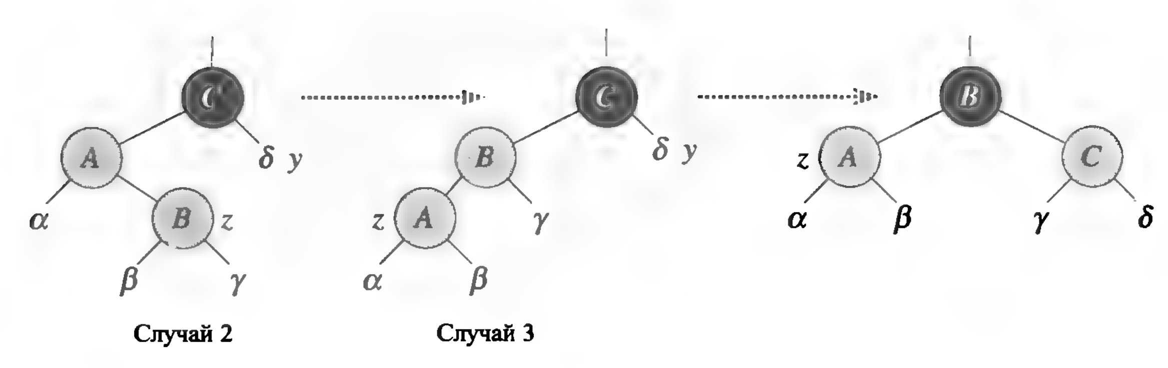


Рис. 13.6. Случаи 2 и 3 процедуры RB-INSERT-FIXUP. Как и в случае 1, свойство 4 нарушается либо случаем 2, либо случаем 3, поскольку z и его родительский узел $z.p$ красные. Каждое из поддеревьев α, β, γ и δ имеет черный корень (α, β и γ согласно свойству 4, а δ — поскольку в противном случае мы получили бы случай 1), и каждое из них имеет одну и ту же черную высоту. Мы преобразуем случай 2 в случай 3 левым поворотом, который сохраняет свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Случай 3 приводит к определенному перекрашиванию и правому повороту, что также сохраняет свойство 5. Затем цикл **while** завершается, поскольку свойство 4 удовлетворено: в одной строке больше нет двух красных узлов.

псевдокода соответствуют случаю 2, который показан на рис. 13.6 вместе со случаем 3. В случае 2 узел z является правым потомком своего родительского узла. Мы используем левый поворот для преобразования сложившейся ситуации в случай 3 (строки 12–14), когда z является левым потомком. Поскольку и z, и $z.p$ — красные узлы, поворот не влияет ни на черную высоту узлов, ни на выполнение свойства 5. Когда мы приходим к случаю 3 (либо непосредственно, либо поворотом из случая 2), узел y, дядя узла z, имеет черный цвет (поскольку иначе мы бы получили случай 1). Кроме того, обязательно существует узел $z.p.p$, так как мы доказали, что этот узел существовал при выполнении строк 2 и 3, а также что после перемещения узла z на один узел вверх в строке 10 с последующим опусканием на один уровень в строке 11 узел $z.p.p$ остается неизменным. В случае 3 мы выполняем ряд изменений цвета и правый поворот, которые сохраняют свойство 5. После этого, так как у нас нет двух идущих подряд красных узлов, работа процедуры завершается. Больше тело цикла **while** не выполняется, так как узел $z.p$ теперь черный.

Покажем, что случаи 2 и 3 сохраняют инвариант цикла. (Как мы только что доказали, перед следующей проверкой в строке 1 узел $z.p$ будет черным и тело цикла больше выполняться не будет.)

а. В случае 2 выполняется присвоение, после которого z указывает на красный узел $z.p$. Никаких других изменений z или его цвета в случаях 2 и 3 не выполняется.

б. В случае 3 узел $z.p$ делается черным, так что если $z.p$ в начале следующей итерации является корнем, то этот корень — черный.

в. Как и в случае 1, в случаях 2 и 3 свойства 1, 3 и 5 сохраняются.

Поскольку узел z в случаях 2 и 3 не является корнем, нарушение свойства 2 невозможно. Случаи 2 и 3 не могут приводить к нарушению свойства 2.

поскольку при повороте в случае 3 сделанный красным узел становится дочерним по отношению к черному.

Таким образом, случаи 2 и 3 приводят к коррекции нарушения свойства 4, при этом не внося никаких новых нарушений красно-черных свойств.

Показав, что при любой итерации инвариант цикла сохраняется, мы тем самым показали, что процедура RB-INSERT-FIXUP корректно восстанавливает красно-черные свойства дерева.

Анализ

Чему равно время работы процедуры RB-INSERT? Поскольку высота красно-черного дерева с n узлами равна $O(\lg n)$, выполнение строк 1–16 процедуры RB-INSERT требует времени $O(\lg n)$. В процедуре RB-INSERT-FIXUP цикл **while** повторно выполняется только в случае 1, и указатель z при этом перемещается вверх по дереву на два уровня. Таким образом, общее количество возможных выполнений тела цикла **while** равно $O(\lg n)$. Следовательно, общее время работы процедуры RB-INSERT равно $O(\lg n)$. Интересно, что в ней никогда не выполняется больше двух поворотов, поскольку цикл **while** в случаях 2 и 3 завершает работу.

Упражнения

13.3.1

В строке 16 процедуры RB-INSERT мы окрашиваем вновь вставленный узел в красный цвет. Заметим, что если бы мы окрашивали его в черный цвет, то свойство 4 красно-черного дерева не было бы нарушено. Так почему же мы не делаем этого?

13.3.2

Изобразите красно-черные деревья, которые образуются при последовательной вставке ключей $41, 38, 31, 12, 19, 8$ в изначально пустое красно-черное дерево.

13.3.3

Предположим, что черная высота каждого из поддеревьев α, β, γ, δ и ε на рис. 13.5 и 13.6 равна k. Найдите черные высоты каждого узла на этих рисунках, чтобы убедиться, что свойство 5 сохраняется при указанных преобразованиях.

13.3.4

Профессор озабочен вопросом, не может ли в процедуре RB-INSERT-FIXUP произойти присвоение значения RED узлу $T.nil.color$, ведь в этом случае проверка в строке 1 не вызовет окончания работы цикла, если z — корень дерева. Покажите, что страхи профессора безосновательны, доказав, что процедура RB-INSERT-FIXUP никогда не окрашивает $T.nil.color$ в красный цвет.

13.3.5

Рассмотрим красно-черное дерево, образованное вставкой n узлов с помощью процедуры RB-INSERT. Докажите, что если $n > 1$, то в дереве имеется как минимум один красный узел.

13.3.6

Предложите эффективную реализацию процедуры RB-INSERT в случае, когда представление красно-черных деревьев не включает указатель на родительский узел.

13.4. Удаление

Как и остальные базовые операции над красно-черными деревьями с n узлами, удаление узла выполняется за время $O(\lg n)$. Удаление оказывается несколько более сложной задачей, чем вставка.

Процедура для удаления узла из красно-черного дерева основана на процедуре TREE-DELETE (раздел 12.3). Сначала нужно внести изменения в подпрограмму TRANSPLANT, которую процедура TREE-DELETE вызывает в процессе работы с красно-черным деревом.

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

Процедура RB-TRANSPLANT имеет два отличия от процедуры TRANSPLANT. Во-первых, строка 1 обращается к ограничителю $T.nil$, а не к NIL. Во-вторых, присваивание атрибуту $v.p$ в строке 6 выполняется безусловно: возможно выполнение присваивания, даже если v указывает на ограничитель. В действительности мы будем использовать возможность присваивания атрибуту $v.p$ при $v = T.nil$.

Процедура RB-DELETE подобна процедуре TREE-DELETE, но имеет дополнительные строки псевдокода. Некоторые из них отслеживают узел y, который может вызвать нарушения красно-черных свойств. Если нужно удалить узел z и z имеет меньше двух дочерних узлов, то z удаляется из дерева, и мы делаем y совпадающим с z. Если у z два дочерних узла, то узел y должен быть преемником z в дереве, и y перемещается в дереве в позицию узла z. Мы также запоминаем цвет y перед его удалением или перемещением и отслеживаем узел x, который перемещается в исходную позицию узла y в дереве, поскольку узел x также может привести к нарушению красно-черных свойств. После удаления узла z процедура RB-DELETE вызывает вспомогательную процедуру RB-DELETE-FIXUP, которая

изменяет цвета и выполняет повороты для восстановления свойств красно-черного дерева.

RB-DELETE(T, z)

```
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
7      x = z.left
8      RB-TRANSPLANT(T, z, z.left)
9  else y = TREE-MINIMUM(z.right)
10     y-original-color = y.color
11     x = y.right
12     if y.p == z
13         x.p = y
14     else RB-TRANSPLANT(T, y, y.right)
15         y.right = z.right
16         y.right.p = y
17     RB-TRANSPLANT(T, z, y)
18     y.left = z.left
19     y.left.p = y
20     y.color = z.color
21 if y-original-color == BLACK
22     RB-DELETE-FIXUP(T, x)
```

Хотя процедура RB-DELETE содержит почти в два раза больше строк, чем псевдокод TREE-DELETE, обе эти процедуры имеют одинаковую базовую структуру. Каждую строку TREE-DELETE можно найти в RB-DELETE (с тем отличием, что NIL заменено $T.nil$, а вызов TRANSPLANT — вызовом RB-TRANSPLANT), и выполняются эти строки при одних и тех же условиях.

А вот отличия между этими двумя процедурами.

- Мы поддерживаем узел y в качестве узла, либо удаленного из дерева, либо перемещенного в пределах последнего. В строке 1 y становится указывающим на узел z, если z имеет меньше двух дочерних узлов и, таким образом, оказывается удаленным. Когда z имеет два дочерних узла, в строке 9 y становится указывающим на узел, следующий в дереве за z, так же, как в процедуре TREE-DELETE, и y перемещается в дереве в позицию узла z.
- Поскольку цвет узла y может измениться, переменная *y-original-color* хранит цвет узла y до любых изменений цвета. В строках 2 и 10 выполняется установка этой переменной немедленно после присваивания значения переменной y. Когда z имеет два дочерних узла, $y \neq z$ и узел y перемещается в исходную позицию узла z в красно-черном дереве; строка 20 назначает y тот же цвет, что

и цвет узла z. Необходимо хранить исходный цвет y для его проверки в конце процедуры RB-DELETE; если он был черным, то удаление или перемещение y может привести к нарушениям свойств красно-черного дерева.

- Как уже говорилось, мы отслеживаем узел x, который перемещается в исходную позицию узла y. Присваивания в строках 4, 7 и 11 делают x указывающим либо на единственный дочерний узел узла y, либо, если y не имеет дочерних узлов, на ограничитель $T.nil$. (Вспомним из раздела 12.3, что у узла y нет левого дочернего узла.)

- Поскольку узел x перемещается в исходную позицию узла y, атрибут $x.p$ всегда устанавливается указывающим на исходную позицию родительского по отношению к y узла в дереве, даже если x в действительности является ограничителем $T.nil$. Присваивание атрибуту $x.p$ в строке 6 процедуры RB-TRANSPLANT имеет место во всех случаях, кроме ситуации, когда z исходно является родителем y (что осуществляется, только когда z имеет два дочерних узла и следующий за z элемент y представляет собой правый дочерний узел z). (Заметим, что когда RB-TRANSPLANT вызывается в строке 5, 8 или 14, третий передаваемый параметр совпадает с x.)

  Однако если исходным родительским узлом узла y является узел z, нам не нужно, чтобы атрибут $x.p$ указывал на исходный родитель y, поскольку мы удаляем этот узел из дерева. Поскольку узел y передвинется вверх и займет в дереве позицию узла z, установка $x.p$ равным y в строке 13 приведет к тому, что $x.p$ будет указывать на исходную позицию родителя y, даже если $x = T.nil$.

- Наконец, если узел y был черным, в свойства красно-черного дерева может быть внесено одно или несколько нарушений, так что для восстановления свойств красно-черного дерева в строке 22 выполняется вызов RB-DELETE-FIXUP. Если узел y был красным, при переименовании или удалении узла y красно-черные свойства сохраняются по следующим причинам.

  1. Ни одна черная высота в дереве не меняется.
  2. Никакие красные узлы не делаются смежными. Поскольку y занимает в дереве место z вместе с цветом узла z, мы не можем получить два смежных красных узла в новой позиции узла y в дереве. Кроме того, если y не был правым дочерним узлом z, исходный правый дочерний узел x узла y заменяет последний в дереве. Если y красный, то x должен быть черным, так что замена y на x не может привести к тому, что два красных узла станут смежными.
  3. Поскольку узел y не может быть корнем, если он был красным, корень остается черным.

Если узел y был черным, то могут возникнуть три проблемы, которые исправит вызов RB-DELETE-FIXUP. Во-первых, если y был корнем, а теперь новым корнем стал красный потомок y, нарушается свойство 2. Во-вторых, если и x, и $x.p$ красные, то нарушается свойство 4. И в-третьих, перемещение y в дереве

приводит к тому, что любой простой путь, ранее содержавший y, теперь имеет на один черный узел меньше. Таким образом, для всех предков y оказывается нарушенным свойство 5. Мы можем исправить ситуацию, утверждая, что узел x, ныне занимающий исходную позицию y, — "сверхчерный", т.е. при рассмотрении любого простого пути, проходящего через x, следует добавлять дополнительную единицу к количеству черных узлов. При такой интерпретации свойство 5 остается выполняющимся. При удалении или перемещении черного узла y мы передаем его "черноту" узлу x. Проблема заключается в том, что теперь узел x не является ни черным, ни красным, что нарушает свойство 1. Вместо этого узел x окрашен либо "дважды черным", либо "красно-черным" цветом, что дает при подсчете черных узлов на простых путях, содержащих x, вклад, равный соответственно 2 или 1. Атрибут *color* узла x при этом остается равным либо RED (если узел красно-черный), либо BLACK (если узел дважды черный). Другими словами, цвет узла x не соответствует его атрибуту *color*.

Теперь рассмотрим процедуру RB-DELETE-FIXUP и то, как она восстанавливает красно-черные свойства дерева поиска.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root и x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                          // Случай 1
            x.p.color = RED                          // Случай 1
            LEFT-ROTATE(T, x.p)                      // Случай 1
            w = x.p.right                            // Случай 1
        if w.left.color == BLACK и w.right.color == BLACK
            w.color = RED                            // Случай 2
            x = x.p                                  // Случай 2
        else if w.right.color == BLACK
                w.left.color = BLACK n               // Случай 3
                nw.color = RED                       // Случай 3
                RIGHT-ROTATE(T, w)                   // Случай 3
                w = x.p.right                        // Случай 3
            w.color = x.p.color                      // Случай 4
            x.p.color = BLACK                        // Случай 4
            w.right.color = BLACK                    // Случай 4
            LEFT-ROTATE(T, x.p)                      // Случай 4
            x = T.root                               // Случай 4
    else (то же, что и в части then, но с заменой
            "правого" (right) "левым" (left) и наоборот)
x.color = BLACK
```

Процедура RB-DELETE-FIXUP восстанавливает свойства 1, 2 и 4. В упр. 13.4.1 и 13.4.2 требуется показать, что эта процедура восстанавливает свойства 2 и 4,

так что в оставшейся части раздела мы обратим свое внимание на свойство 1. Цель цикла **while** в строках 1–22 заключается в перенесении дополнительной "черноты" вверх по дереву до тех пор, пока не выполнится одно из следующих условий.

1. x указывает на красно-черный узел — в этом случае мы просто делаем узел x "единожды черным" в строке 23.
2. x указывает на корень — в этом случае мы просто убираем излишнюю черноту.
3. Выполнив некоторые повороты и перекраску, мы выходим из цикла.

Внутри цикла **while** x всегда указывает на дважды черный узел, не являющийся корнем. В строке 2 мы определяем, является ли x левым или правым дочерним узлом своего родителя $x.p$. (Приведен подробный код для ситуации, когда x — левый потомок. Для правого потомка код аналогичен, симметричен и скрыт за описанием в строке 22.) Поддерживается указатель w, который указывает на второй потомок родителя x. Поскольку узел x дважды черный, узел w не может быть $T.nil$; в противном случае количество черных узлов на простом пути от $x.p$ к (единожды черному) листу w было бы меньше, чем количество черных узлов на простом пути от $x.p$ к x.

Четыре разных возможных случая[2] показаны на рис. 13.7. Перед тем как приступить к детальному рассмотрению каждого случая, убедимся, что в каждом из случаев преобразования сохраняется свойство 5. Ключевая идея заключается в необходимости убедиться, что применяемые преобразования в каждом случае сохраняют количество черных узлов (включая дополнительную черноту в x) на пути от корня включительно до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$. Таким образом, если свойство 5 выполнялось до преобразования, оно выполняется и после него. Например, на рис. 13.7, (а), который иллюстрирует случай 1, количество черных узлов на пути от корня до поддеревьев α и β равно 3 как до, так и после преобразования (не забудьте о том, что узел x — дважды черный). Аналогично количество черных узлов на пути от корня до любого из поддеревьев γ, δ, ε и ζ равно 2 как до, так и после преобразования. На рис. 13.7, (б) подсчет должен включать значение c, равное значению атрибута *color* корня показанного поддерева, которое может быть либо RED, либо BLACK. Если определить $\text{count}(\text{RED}) = 0$ и $\text{count}(\text{BLACK}) = 1$, то на пути от корня до поддерева α количество черных узлов равно $2 + \text{count}(c)$; эта величина одинакова до и после выполнения преобразований. В такой ситуации после преобразования новый узел x имеет атрибут *color*, равный c, но реально это либо красно-черный узел (если $c = \text{RED}$), либо дважды черный (если $c = \text{BLACK}$). Прочие случаи могут быть проверены аналогично (см. упр. 13.4.5).

---

[2]Как и в процедуре RB-INSERT-FIXUP, случаи в процедуре RB-DELETE-FIXUP не являются взаимоисключающими.

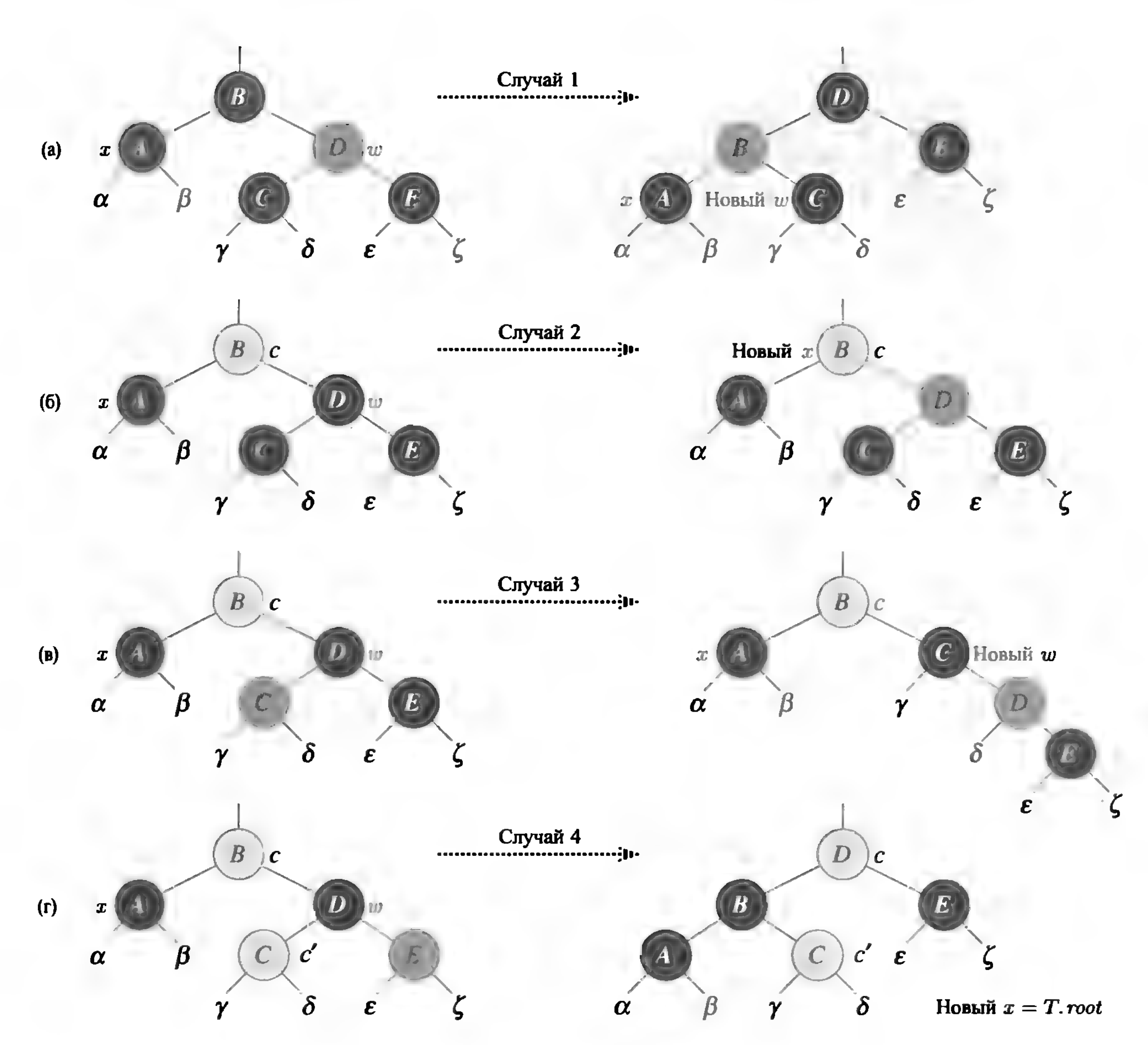


Рис. 13.7. Случаи цикла **while** процедуры RB-DELETE-FIXUP. У черных узлов атрибут *color* равен BLACK, у темно-серых — атрибут *color* равен RED, а у светлых узлов атрибут *color* представлен значениями c и c', которые могут быть либо RED, либо BLACK. Буквы $\alpha, \beta, \ldots, \zeta$ представляют произвольные поддеревья. Каждый случай преобразует конфигурацию, показанную слева, в конфигурацию, показанную справа, путем изменения некоторых цветов и/или поворота. Любой узел, на который указывает x, имеет дополнительный черный цвет и является либо дважды черным, либо красно-черным. Цикл повторяется только в случае 2. **(а)** Случай 1 преобразуется в случай 2, 3 или 4 путем изменения цвета узлов B и D и выполнения левого поворота. **(б)** В случае 2 дополнительная чернота, представленная указателем x, перемещается вверх по дереву путем окраски узла D в красный цвет и установки x указывающим на узел B. Если мы попадаем в случай 2 из случая 1, цикл **while** завершается, поскольку новый узел x — черно-красный, и, следовательно, значение c его атрибута *color* равно RED. **(в)** Случай 3 преобразуется в случай 4 путем обмена цветов узлов C и D и выполнения правого поворота. **(г)** Случай 4 убирает дополнительную черноту, представленную x, путем изменения некоторых цветов и выполнения левого поворота (без нарушения красно-черных свойств), после чего цикл завершается.

Случай 1. Брат w узла x — красный

Случай 1 (строки 5–8 процедуры RB-DELETE-FIXUP и рис. 13.7, (а)) возникает, когда узел w ("брат" узла x) — красный. Поскольку w должен иметь черные потомки, можно обменять цвета w и $x.p$, а затем выполнить левый поворот вокруг $x.p$ без нарушения каких-либо красно-черных свойств. Новый "брат" x, до поворота бывший одним из дочерних узлов w, теперь черный. Таким путем случай 1 приводится к случаю 2, 3 или 4.

Случаи 2, 3 и 4 возникают при черном узле w и отличаются один от другого цветами дочерних по отношению к w узлов.

Случай 2. Узел w — черный, оба его дочерних узла — черные

В этом случае (строки 10 и 11 процедуры RB-DELETE-FIXUP и рис. 13.7, (б)) оба дочерних узла w — черные. Поскольку узел w также черный, мы можем забрать черную окраску у x и w, сделав x единожды черным, а w — красным. Для того чтобы компенсировать удаление черной окраски x и w, мы можем добавить дополнительный черный цвет узлу $x.p$, который до этого мог быть как красным, так и черным. После этого будет выполнена следующая итерация цикла, в которой роль x будет играть текущий узел $x.p$. Заметим, что если мы переходим к случаю 2 от случая 1, новый узел x — красно-черный, поскольку исходный узел $x.p$ был красным. Следовательно, значение c атрибута *color* нового узла x равно RED и цикл завершается при проверке условия цикла. После этого новый узел x окрашивается в обычный черный цвет в строке 23.

Случай 3. Брат w узла x — черный, левый дочерний узел узла w — красный, а правый — черный

В этом случае (строки 13–16 процедуры RB-DELETE-FIXUP и рис. 13.7, (в)) узел w — черный, его левый дочерний узел — красный, а правый — черный. Мы можем обменять цвета w и его левого дочернего узла $w.left$, а затем выполнить правый поворот вокруг w без нарушения каких-либо красно-черных свойств. Новым "братом" узла x после этого будет черный узел с красным правым дочерним узлом, и, таким образом, случай 3 приводится к случаю 4.

Случай 4. Брат w узла x черный, а правый дочерний узел узла w красный

В этом случае (строки 17–21 процедуры RB-DELETE-FIXUP и рис. 13.7, (г)) узел w — черный, а его правый дочерний узел — красный. Выполняя обмен цветов и левый поворот вокруг $x.p$, мы можем устранить излишнюю черноту в x, делая его просто черным, без нарушения каких-либо красно-черных свойств. Присвоение x указателя на корень дерева приводит к завершению работы цикла при проверке условия при следующей итерации.

Анализ

Чему равно время работы процедуры RB-DELETE? Поскольку высота дерева с n узлами равна $O(\lg n)$, общая стоимость процедуры без вызова вспомога-

тельной процедуры RB-DELETE-FIXUP равна $O(\lg n)$. В процедуре RB-DELETE-FIXUP в случаях 1, 3 и 4 завершение работы происходит после выполнения фиксированного числа изменений цвета и не более трех поворотов. Случай 2 — единственный, после которого возможно выполнение очередной итерации цикла **while**, причем указатель x перемещается вверх по дереву не более чем $O(\lg n)$ раз и никакие повороты при этом не выполняются. Таким образом, время работы процедуры RB-DELETE-FIXUP составляет $O(\lg n)$, причем она выполняет не более трех поворотов. Общее время работы процедуры RB-DELETE, само собой разумеется, также равно $O(\lg n)$.

Упражнения

13.4.1

Покажите, что после выполнения процедуры RB-DELETE-FIXUP корень дерева должен быть черным.

13.4.2

Покажите, что если в процедуре RB-DELETE и x, и $x.p$ красные, то при вызове RB-DELETE-FIXUP(T, x) свойство 4 будет восстановлено.

13.4.3

В упр. 13.3.2 вы построили красно-черное дерево, которое является результатом последовательной вставки ключей $41, 38, 31, 12, 19, 8$ в изначально пустое дерево. Покажите теперь красно-черные деревья, которые будут получаться в результате последовательного удаления ключей в порядке $8, 12, 19, 31, 38, 41$.

13.4.4

В каких строках кода процедуры RB-DELETE-FIXUP мы можем обращаться к ограничителю $T.nil$ или изменять его?

13.4.5

Для каждого из случаев, показанных на рис. 13.7, подсчитайте количество черных узлов на пути от корня показанного поддерева до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$ и убедитесь, что оно не меняется при выполнении преобразований. Если узел имеет атрибут *color*, равный c или c', воспользуйтесь символьными обозначениями $\text{count}(c)$ или $\text{count}(c')$.

13.4.6

Профессор озабочен вопросом, не может ли узел $x.p$ не быть черным в начале случая 1 в процедуре RB-DELETE-FIXUP. Если профессор прав, то строки 5 и 6 процедуры ошибочны. Покажите, что в начале случая 1 узел $x.p$ не может не быть черным, так что профессору нечего волноваться.

13.4.7

Предположим, что узел x вставлен в красно-черное дерево с помощью процедуры RB-INSERT, после чего немедленно удален из этого дерева процедурой RB-

DELETE. Будет ли полученное в результате красно-черное дерево таким же, как и исходное? Обоснуйте свой ответ.

Задачи

13.1. Перманентные динамические множества

Иногда полезно сохранять предыдущие версии динамического множества в процессе его обновления. Такие множества называются ***перманентными*** (persistent). Один из способов реализации перманентного множества состоит в копировании всего множества при внесении в него изменений, однако такой подход может существенно замедлить работу программы и вызвать перерасход памяти. Зачастую эту задачу можно решить гораздо более эффективно.

Рассмотрим перманентное множество S с операциями INSERT, DELETE и SEARCH, которое реализуется с использованием бинарных деревьев поиска, как показано на рис. 13.8, (а). Для каждой версии множества мы поддерживаем отдельный корень. Для добавления ключа 5 в множество создается новый узел с ключом 5, который становится левым дочерним узлом нового узла с ключом 7, так как менять существующий узел с ключом 7 мы не можем. Аналогично новый узел с ключом 7 становится левым потомком нового узла с ключом 8, правым потомком которого является существующий узел с ключом 10. Новый узел с ключом 8 становится, в свою очередь, правым потомком нового корня r' с ключом 4, левым потомком которого является существующий узел с ключом 3. Таким образом, мы копируем только часть дерева, а в остальном используем старое дерево, как показано на рис. 13.8, (б).

Предположим, что каждый узел дерева имеет атрибуты key, $left$ и $right$, но не имеет атрибута с указателем на родительский узел (см. также упр. 13.3.6).

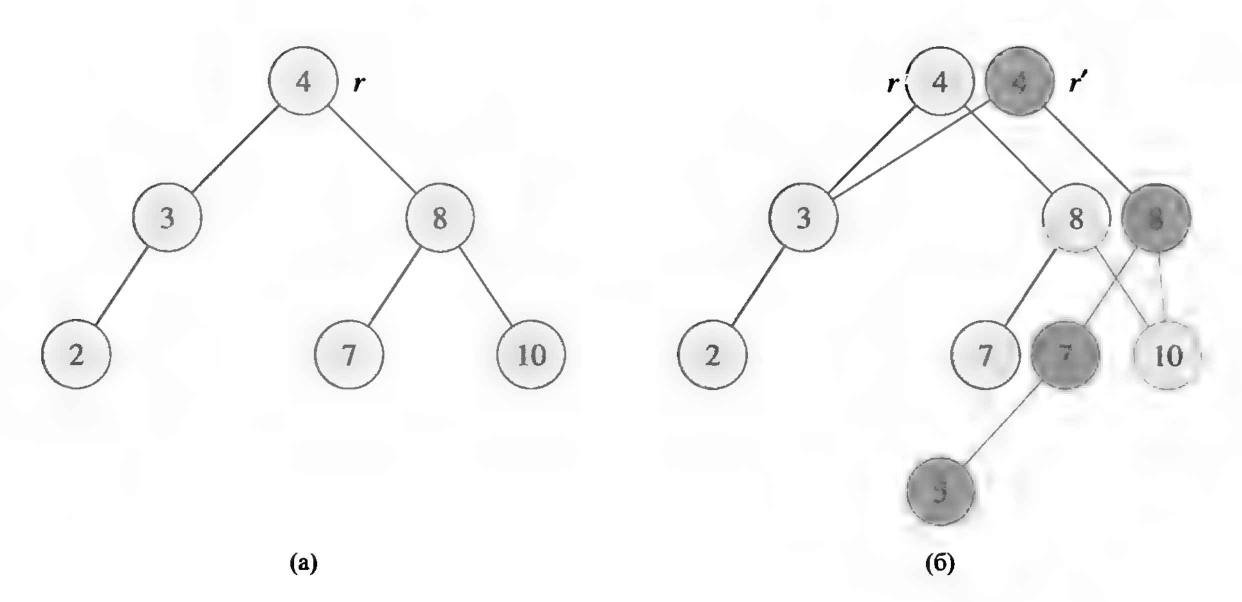


Рис. 13.8. **(а)** Бинарное дерево поиска с ключами 2, 3, 4, 7, 8, 10. **(б)** Перманентное бинарное дерево поиска, полученное в результате вставки ключа 5. Последняя версия множества состоит из узлов, достижимых из корня r', а предыдущая версия состоит из узлов, достижимых из r. Темные узлы добавлены при вставке ключа 5.

а. Определите, какие узлы перманентного бинарного дерева поиска должны быть изменены при вставке в него ключа k или удалении узла y в общем случае.

б. Напишите процедуру PERSISTENT-TREE-INSERT, которая для данного перманентного дерева T и вставляемого ключа k возвращает новое перманентное дерево T', получающееся в результате вставки k в T.

в. Если высота перманентного бинарного дерева поиска T равна h, сколько времени будет работать ваша реализация PERSISTENT-TREE-INSERT и какие требования к памяти она предъявляет? (Количество требуемой памяти пропорционально количеству новых узлов.)

г. Предположим, что в каждом узле имеется атрибут, который представляет собой указатель на родительский узел. В этом случае процедура PERSISTENT-TREE-INSERT должна выполнять дополнительное копирование. Докажите, что в этом случае время работы процедуры и объем необходимой памяти равны $\Omega(n)$, где n — количество узлов в дереве.

д. Покажите, как можно использовать красно-черные деревья для того, чтобы гарантировать при вставке и удалении в наихудшем случае равенство $O(\lg n)$ времени работы процедуры и объема необходимой памяти.

13.2. Операция соединения красно-черных деревьев

Операция ***соединения*** (join) применяется к двум динамическим множествам S_1 и S_2 и элементу x (такому, что для любых $x_1 \in S_1$ и $x_2 \in S_2$ выполняется неравенство $x_1.key \leq x.key \leq x_2.key$). Результатом операции является множество $S = S_1 \cup \{x\} \cup S_2$. В данной задаче мы рассмотрим реализацию операции соединения красно-черных деревьев.

а. Будем хранить черную высоту красно-черного дерева T как новый атрибут $T.bh$. Покажите, что этот атрибут может поддерживаться процедурами RB-INSERT и RB-DELETE без использования дополнительной памяти в узлах дерева и без увеличения асимптотического времени работы процедур. Покажите, что при спуске по дереву T можно определить черную высоту каждого посещаемого узла за время $O(1)$ на каждый посещенный узел.

Мы хотим реализовать операцию RB-JOIN(T_1, x, T_2), которая, разрушая деревья T_1 и T_2, возвращает красно-черное дерево $T = T_1 \cup \{x\} \cup T_2$. Пусть n — общее количество узлов в деревьях T_1 и T_2.

б. Считая, что $T_1.bh \geq T_2.bh$, разработайте алгоритм, который за время $O(\lg n)$ находит в дереве T_1 среди узлов, черная высота которых равна $T_2.bh$, черный узел y с наибольшим значением ключа.

в. Пусть T_y — поддерево с корнем y. Опишите, как заменить T_y на $T_y \cup \{x\} \cup T_2$ за время $O(1)$ с сохранением свойства бинарного дерева поиска.

г. В какой цвет нужно окрасить x, чтобы сохранились красно-черные свойства 1, 3 и 5? Опишите, как восстановить свойства 2 и 4 за время $O(\lg n)$.

д. Докажите, что предположение в п. (б) данной задачи не приводит к потере общности. Опишите симметричную ситуацию, возникающую при $T_1.bh \leq T_2.bh$.

е. Покажите, что время работы процедуры RB-JOIN равно $O(\lg n)$.

13.3. AVL-деревья

AVL-дерево представляет собой бинарное дерево поиска со ***сбалансированной высотой***: для каждого узла x высота левого и правого поддеревьев x отличается не более чем на 1. Для реализации AVL-деревьев мы воспользуемся дополнительным атрибутом $x.h$ в каждом узле дерева, в котором хранится высота данного узла. Как и в случае обычных деревьев поиска, мы считаем, что $T.root$ указывает на корневой узел.

а. Докажите, что AVL-дерево с n узлами имеет высоту $O(\lg n)$. (*Указание:* докажите, что в AVL-дереве высотой h имеется как минимум F_h узлов, где F_h — h-е число Фибоначчи.)

б. Для вставки узла в AVL-дерево он сначала размещается в соответствующем месте бинарного дерева поиска. После этого дерево может оказаться несбалансированным, в частности, высота левого и правого потомков некоторого узла может отличаться на 2. Разработайте процедуру BALANCE(x), которая получает в качестве параметра поддерево с корнем в узле x, левый и правый потомки которого сбалансированы по высоте и имеют высоту, отличающуюся не более чем на 2 (т.е. $|x.right.h - x.left.h| \leq 2$), и изменяет его таким образом, что поддерево с корнем в узле x становится сбалансированным по высоте. (*Указание:* воспользуйтесь поворотами.)

в. Используя решение подзадачи (б), разработайте рекурсивную процедуру AVL-INSERT(x, z), которая получает в качестве параметров узел x в AVL-дереве и вновь созданный узел z (с заполненным полем ключа) и вставляет z в поддерево, корнем которого является узел x, сохраняя при этом свойство, заключающееся в том, что x — корень AVL-дерева. Как и в случае процедуры TREE-INSERT из раздела 12.3, считаем, что атрибут $z.key$ заполнен и что $z.left = \text{NIL}$ и $z.right = \text{NIL}$. Кроме того, полагаем, что $z.h = 0$. Таким образом, для вставки узла z в AVL-дерево T мы должны осуществить вызов AVL-INSERT($T.root, z$).

г. Покажите, что время работы операции AVL-INSERT для AVL-дерева с n узлами равно $O(\lg n)$ и она выполняет $O(1)$ поворотов.

13.4. Дерамиды[3]

Если мы вставляем в бинарное дерево поиска набор из n элементов, то полученное в результате дерево может оказаться ужасно несбалансированным, что приводит к большому времени поиска. Однако, как мы видели в разделе 12.4, случайные бинарные деревья поиска обычно оказываются достаточно сбалансированными. Таким образом, стратегия построения сбалансированного дерева для фиксированного множества элементов состоит в их случайной перестановке с последующей вставкой в дерево.

Но что делать, если все элементы недоступны одновременно? Если мы получаем элементы по одному, можем ли мы построить из них случайное бинарное дерево поиска?

Рассмотрим структуру данных, которая позволяет положительно ответить на этот вопрос. ***Дерамида*** представляет собой бинарное дерево поиска с модифицированным способом упорядочения узлов. Пример дерамиды показан на рис. 13.9. Как обычно, каждый узел x в дереве имеет значение ключа $x.key$. Кроме того, мы назначим каждому узлу атрибут $x.priority$, который представляет собой случайное число, выбираемое для каждого узла независимо от других. Мы считаем, что все приоритеты и все ключи в дереве различны. Узлы дерамиды упорядочены таким образом, чтобы ключи подчинялись свойству бинарных деревьев поиска, а приоритеты — свойству неубывающей пирамиды.

- Если v является левым дочерним узлом узла u, то $v.key < u.key$.
- Если v является правым дочерним узлом узла u, то $v.key > u.key$.
- Если v является дочерним узлом узла u, то $v.priority > u.priority$.

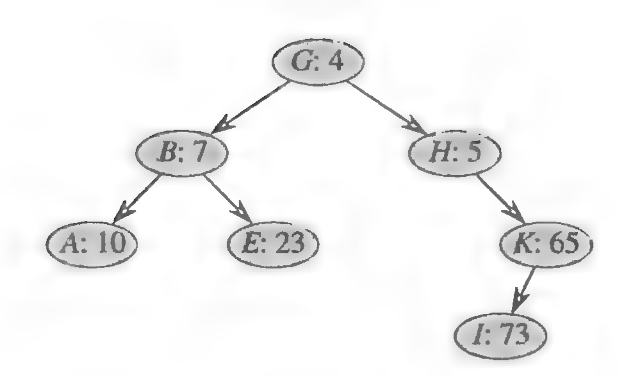


Рис. 13.9. Дерамида. Каждый узел x помечен атрибутами $x.key$: $x.priority$. Например, корень имеет ключ G и приоритет 4.

Именно эта комбинация свойств дерева и пирамиды и дала название "дерамида" (treap).

Помочь разобраться в дерамидах может следующая интерпретация. Предположим, что мы вставляем в дерамиду узлы $x_1, x_2, \ldots, x_n$ со связанными с ни-

[3] В оригинале — "treap"; слово, образованное из двух — "tree" и "heap". Аналогично из слов "дерево" и "пирамида" получился русский эквивалент. — *Примеч. пер.*

ми ключами. Тогда полученная в результате дерамида представляет собой дерево, которое получилось бы в результате вставки узлов в обычное бинарное дерево поиска в порядке, определяемом (случайно выбранными) приоритетами, т.е. $x_i.priority < x_j.priority$ означает, что узел x_i был вставлен до узла x_j.

а. Покажите, что для каждого заданного множества узлов $x_1, x_2, \ldots, x_n$ со связанными с ними ключами и приоритетами (все ключи и приоритеты различны) существует единственная дерамида.

б. Покажите, что математическое ожидание высоты дерамиды равно $\Theta(\lg n)$ и, следовательно, время поиска заданного значения в дерамиде составляет $\Theta(\lg n)$.

Рассмотрим процесс вставки нового узла в существующую дерамиду. Первое, что мы делаем, — назначаем новому узлу случайное значение приоритета. Затем мы вызываем процедуру вставки, названную нами TREAP-INSERT, работа которой показана на рис. 13.10.

в. Объясните, как работает процедура TREAP-INSERT. Поясните принцип ее работы обычным русским языком и приведите ее псевдокод. (*Указание:* выполните обычную вставку в бинарное дерево поиска, а затем выполните повороты для восстановления свойства неубывающей пирамиды.)

г. Покажите, что ожидаемое время работы процедуры TREAP-INSERT составляет $\Theta(\lg n)$.

Процедура TREAP-INSERT выполняет поиск с последующей последовательностью поворотов. Хотя эти две операции имеют одно и то же ожидаемое время работы, на практике стоимость этих операций различна. Поиск считывает информацию из дерамиды, никак ее не изменяя. Повороты, напротив, приводят к изменению указателей на дочерние и родительские узлы в дерамиде. На большинстве компьютеров операция чтения существенно более быстрая, чем операция записи. Соответственно, желательно, чтобы поворотов при выполнении процедуры TREAP-INSERT было как можно меньше. Мы покажем, что ожидаемое количество выполняемых процедурой поворотов ограничено константой.

Для этого необходимо дать несколько определений, проиллюстрированных на рис. 13.11. ***Левый хребет*** бинарного дерева поиска T представляет собой простой путь от корня к узлу с минимальным значением ключа. Другими словами, левый хребет представляет собой простой путь, состоящий только из левых ребер. Соответственно, ***правый хребет*** представляет собой простой путь, состоящий только из правых ребер. Длина хребта — это количество составляющих его узлов.

д. Рассмотрите дерамиду T непосредственно после того, как в нее с помощью процедуры TREAP-INSERT вставляется узел x. Пусть C — длина правого хребта левого поддерева x, а D — длина левого хребта правого поддерева x. Докажите, что общее количество поворотов, которые были выполнены в процессе вставки x, равно $C + D$.

Рис. 13.10. Работа процедуры TREAP-INSERT. **(а)** Исходная дерамида до вставки. **(б)** Дерамида после вставки узла с ключом C и приоритетом 25. **(в)–(г)** Промежуточные стадии при вставке узла с ключом D и приоритетом 9. **(д)** Дерамида после выполнения вставки в частях (в) и (г). **(е)** Дерамида после вставки узла с ключом F и приоритетом 2.

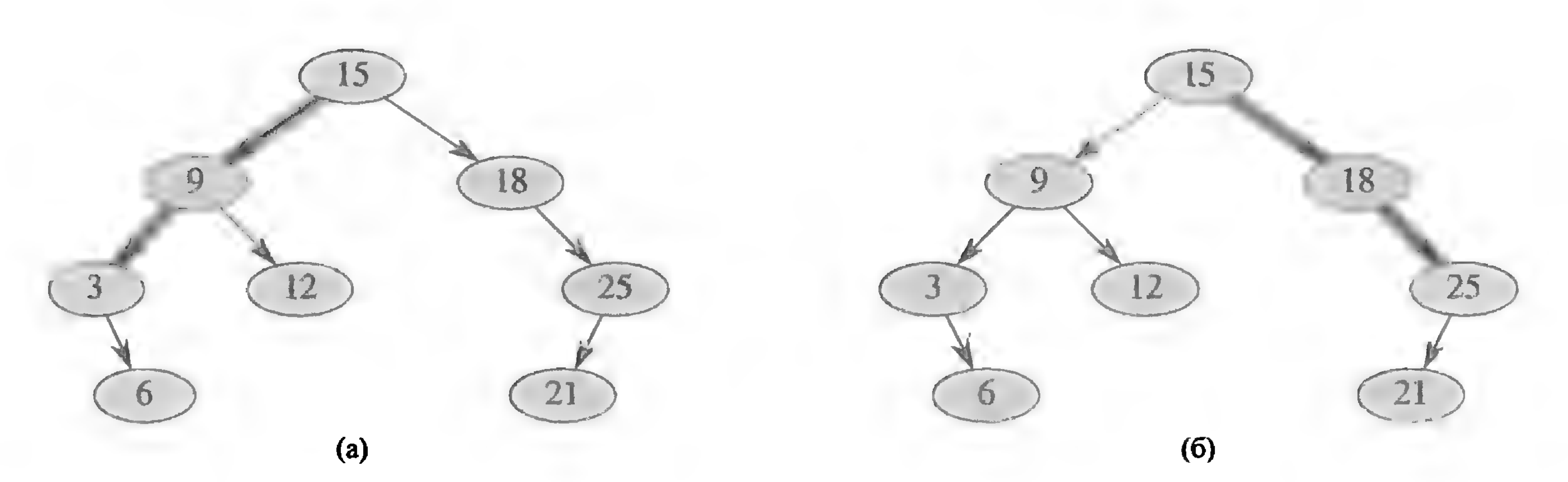


Рис. 13.11. Хребты бинарного дерева поиска. Левый хребет выделен в части **(а)**, правый — в части **(b)**.

Теперь мы вычислим математическое ожидание значений C и D. Без потери общности будем считать, что ключи представляют собой натуральные числа $1, 2, \ldots, n$, поскольку мы сравниваем их только друг с другом.

Пусть для узлов x и y ($y \neq x$) дерамиды T $k = x.key$ и $i = y.key$. Определим индикаторную случайную величину

$$X_{ik} = \mathrm{I}\{y \text{ находится в правом хребте левого поддерева } x\} .$$

е. Покажите, что $X_{ik} = 1$ тогда и только тогда, когда $y.priority > x.priority$, $y.key < x.key$, а для каждого z, такого, что $y.key < z.key < x.key$, мы имеем $y.priority < z.priority$.

ж. Покажите, что

$$\begin{aligned}\Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)} .\end{aligned}$$

з. Покажите, что

$$\begin{aligned}\mathrm{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k} .\end{aligned}$$

и. Воспользуйтесь симметрией, чтобы показать, что

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1} .$$

к. Сделайте вывод о том, что математическое ожидание количества поворотов, выполняемых при вставке узла в дерамиду, меньше 2.

Заключительные замечания

Идея балансировки деревьев поиска принадлежит советским математикам Г.М. Адельсону-Вельскому и Е.М. Ландису [2], которые в 1962 году предложили класс сбалансированных деревьев поиска, получивших название "AVL-деревья" (описаны в задаче 13.3). Еще один класс деревьев поиска, называемых 2-3-деревьями, был предложен Д.Э. Хопкрофтом (J.E. Hopcroft, не опубликовано) в 1970 году. Баланс этих деревьев поддерживается с помощью изменения степеней ветвления в узлах. Обобщение 2-3-деревьев, предложенное Байером

(Bayer) и Мак-Крейтом (McCreight) [34] под названием "В-деревья", рассматривается в главе 18.

Красно-черные деревья были предложены Байером [33] под названием "симметричные бинарные В-деревья". Гибас (Guibas) и Седжвик (Sedgewick) [154] детально изучили их свойства и предложили использовать концепцию красного и черного цветов. Андерссон (Andersson) [15] предложил вариант красно-черных деревьев, обладающих повышенной простотой кодирования (который был назван Вейссом (Weiss) [349] АА-деревьями). Эти деревья подобны красно-черным деревьям с тем отличием, что левый потомок в них не может быть красным.

Дерамиды, рассмотренные в задаче 13.4, были предложены Сиделем (Siedel) и Арагоном (Aragon) [307]. Они представляют собой реализацию словаря по умолчанию в LEDA [251], тщательно разработанном наборе структур данных и алгоритмов.

Имеется множество других вариантов сбалансированных бинарных деревьев, включая взвешенно-сбалансированные деревья [262], деревья с k соседями [243], так называемые "деревья — козлы отпущения" (scapegoat trees) [126] и др. Возможно, наиболее интересны "косые" деревья (splay trees), разработанные Слитором (Sleator) и Таржаном (Tarjan) [318] и обладающие свойством саморегуляции (хорошее описание косых деревьев можно найти в работе Таржана [328]). Косые деревья поддерживают сбалансированность без использования дополнительных условий балансировки типа цветов. Вместо этого всякий раз при обращении над ним выполняются "косые" операции (включающие, в частности, повороты). Амортизированная стоимость (см. главу 17) таких операций в дереве с n узлами составляет $O(\lg n)$.

Альтернативой сбалансированным бинарным деревьям поиска являются списки с пропусками (skip list) [284], которые представляют собой связанные списки, оснащенные рядом дополнительных указателей. Все словарные операции в таких списках с n элементами имеют ожидаемое время выполнения, равное $O(\lg n)$.

Глава 14. Расширение структур данных

Зачастую на практике возникают ситуации, когда "классических" структур данных — таких, как дважды связанные списки, хеш-таблицы или бинарные деревья поиска — оказывается недостаточно для решения поставленных задач. Однако только в крайне редких ситуациях приходится изобретать совершенно новые структуры данных; как правило, достаточно расширить существующую структуру путем хранения в ней дополнительной информации, что позволяет запрограммировать необходимую для данного приложения функциональность. Однако такое расширение структур данных — далеко не всегда простая задача, в первую очередь, из-за необходимости обновления и поддержки дополнительной информации стандартными операциями над структурой данных.

В этой главе рассматриваются две структуры данных, которые построены путем расширения красно-черных деревьев. В разделе 14.1 описывается структура, которая обеспечивает операции поиска порядковых статистик в динамическом множестве. С ее помощью мы можем быстро найти i-е по порядку наименьшее число или ранг данного элемента в упорядоченном множестве. В разделе 14.2 рассматривается общая схема расширения структур данных и доказывается теорема, которая может упростить процесс расширения красно-черных деревьев. В разделе 14.3 эта теорема используется при разработке структуры данных, поддерживающей динамическое множество промежутков, например промежутков времени. Такая структура позволяет быстро находить в множестве промежуток, перекрывающийся с данным.

14.1. Динамические порядковые статистики

В главе 9 было введено понятие порядковой статистики. В частности, i-й порядковой статистикой множества из n элементов ($i \in \{1, 2, \ldots, n\}$) является элемент множества с i-м в порядке возрастания ключом. Мы видели, что любая порядковая статистика может быть найдена в неупорядоченном множестве за время $O(n)$. В этом разделе вы увидите, каким образом можно изменить красно-черные деревья для того, чтобы находить порядковую статистику за время $O(\lg n)$. Вы также узнаете, каким образом можно находить ***ранг*** элемента — его порядковый номер в линейно упорядоченном множестве — за то же время $O(\lg n)$.

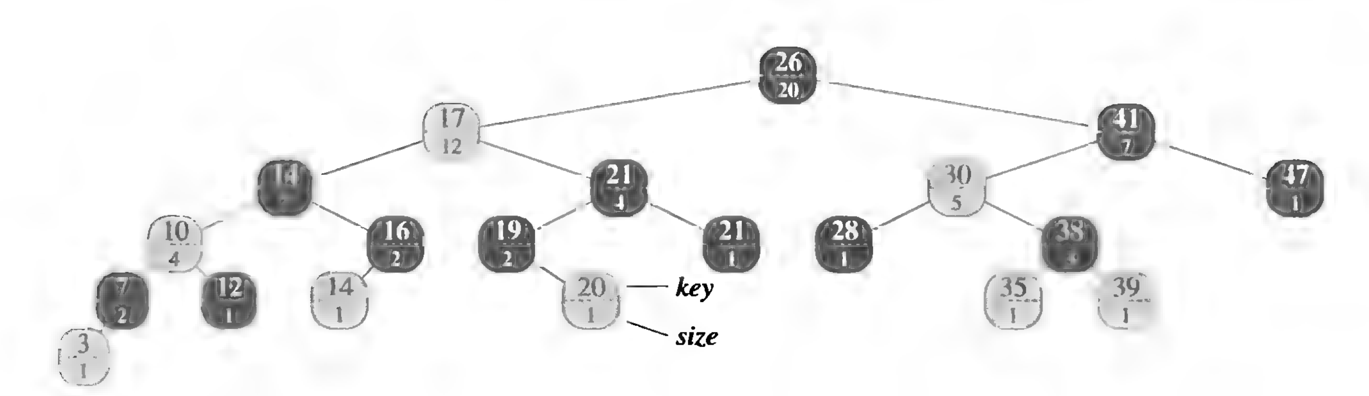


Рис. 14.1. Дерево порядковой статистики, являющееся расширением красно-черного дерева. Светлые узлы — красные, темные — черные. Помимо обычных атрибутов, каждый узел x имеет атрибут $x.size$, который представляет собой количество узлов, отличных от ограничителя, в поддереве с корнем x.

На рис. 14.1 показана структура данных, которая поддерживает быстрые операции порядковой статистики. ***Дерево порядковой статистики*** T (order-statistic tree) представляет собой просто красно-черное дерево с дополнительной информацией, хранящейся в каждом узле. Помимо обычных атрибутов узлов красно-черного дерева $x.key$, $x.color$, $x.p$, $x.left$ и $x.right$, у каждого узла дерева порядковой статистики имеется атрибут $x.size$. Этот атрибут содержит количество внутренних узлов в поддереве с корневым узлом x (включая сам x), т.е. размер поддерева. Если мы определим размер ограничителя как 0, т.е. $T.nil.size = 0$, то получим тождество

$$x.size = x.left.size + x.right.size + 1 .$$

В дереве порядковой статистики условие различности всех ключей не ставится. Например, на рис. 14.1 имеются два ключа со значением 14, и два — со значением 21. При наличии одинаковых ключей определение ранга оказывается нечетким, и мы устраняем неоднозначность дерева порядковой статистики, определяя ранг элемента как позицию, в которой будет выведен данный элемент при центрированном обходе дерева. Например, на рис. 14.1 ключ 14, хранящийся в черном узле, имеет ранг 5, а в красном — ранг 6.

Выборка элемента с заданным рангом

Перед тем как будет изучен вопрос об обновлении информации о размере поддеревьев в процессе вставки и удаления, давайте посмотрим на реализацию двух запросов порядковой статистики, которые используют дополнительную информацию. Начнем с операции поиска элемента с заданным рангом. Процедура OS-SELECT(x, i) возвращает указатель на узел, содержащий i-й в порядке возрастания ключ в поддереве, корнем которого является x (так что для поиска i-го в порядке возрастания ключа в дереве порядковой статистики T мы вызываем процедуру как OS-SELECT$(T.root, i)$).

```
OS-SELECT(x, i)
1  r = x.left.size + 1
2  if i == r
3      return x
4  elseif i < r
5      return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i - r)
```

В строке 1 псевдокода процедуры OS-SELECT мы вычисляем r — ранг узла x в поддереве, для которого он является корнем. Значение $x.left.size$ представляет собой количество узлов, которые идут до x в центрированном обходе поддерева с корнем x. Таким образом, $x.left.size + 1$ является рангом x в поддереве с корнем x. Если $i = r$, то узел x является i-м в порядке возрастания элементом, и мы возвращаем его в строке 3. Если же $i < r$, то i-й в порядке возрастания элемент находится в левом поддереве, так что мы рекурсивно ищем его в поддереве $x.left$ в строке 5. Если же $i > r$, то искомый элемент находится в правом поддереве, и мы делаем соответствующий рекурсивный вызов в строке 6 с учетом того, что i-й в порядке возрастания в дереве с корнем x элемент является $(i - r)$-м в порядке возрастания в правом поддереве x с корнем в $x.right$.

Для того чтобы увидеть описанную процедуру в действии, рассмотрим поиск 17-го в порядке возрастания элемента в дереве порядковой статистики на рис. 14.1. Мы начинаем поиск с корневого узла, ключ которого равен 26, с $i = 17$. Поскольку размер левого поддерева элемента с ключом 26 равен 12, ранг самого́ элемента — 13. Теперь мы знаем, что элемент с рангом 17 является $17 - 13 = 4$-м в порядке возрастания элементом в правом поддереве элемента с ключом 26. После соответствующего рекурсивного вызова x становится узлом с ключом 41, а $i = 4$. Поскольку размер левого поддерева узла с ключом 41 равен 5, ранг этого узла в поддереве равен 6. Теперь мы знаем, что искомый узел находится в левом поддереве узла с ключом 41 и его номер в порядке возрастания — 4. После очередного рекурсивного вызова x становится элементом с ключом 30, а его ранг — 2, и мы рекурсивно ищем элемент с рангом $4 - 2 = 2$ в поддереве, корнем которого является узел с ключом 38. Размер его левого поддерева равен 1, так что ранг самого́ узла с ключом 38 равен 2, и это и есть наш искомый элемент, указатель на который и возвращает процедура.

Поскольку каждый рекурсивный вызов опускает нас на один уровень в дереве порядковой статистики, общее время работы процедуры OS-SELECT в наихудшем случае пропорционально высоте дерева. Поскольку рассматриваемое нами дерево порядковой статистики является красно-черным, его высота равна $O(\lg n)$, где n — количество узлов в дереве. Следовательно, время работы процедуры OS-SELECT в динамическом множестве из n элементов равно $O(\lg n)$.

Определение ранга элемента

Процедура OS-RANK, псевдокод которой приведен далее, по заданному указателю на узел x дерева порядковой статистики T возвращает позицию данного узла при центрированном обходе дерева.

```
OS-RANK(T, x)
1  r = x.left.size + 1
2  y = x
3  while y ≠ T.root
4      if y == y.p.right
5          r = r + y.p.left.size + 1
6      y = y.p
7  return r
```

Процедура работает следующим образом. Ранг x можно рассматривать как число узлов, предшествующих x при центрированном обходе дерева, плюс 1 для самого́ узла x. Процедура OS-RANK поддерживает следующий инвариант цикла.

> В начале каждой итерации цикла **while** в строках 3–6 r представляет собой ранг $x.key$ в поддереве, корнем которого является узел y.

Мы воспользуемся этим инвариантом для того, чтобы показать корректность работы процедуры OS-RANK.

Инициализация. Перед первой итерацией строка 1 устанавливает r равным рангу $x.key$ в поддереве с корнем x. Присвоение $y = x$ в строке 2 делает инвариант истинным при первом выполнении проверки в строке 3.

Сохранение. В конце каждой итерации цикла **while** выполняется присвоение $y = y.p$. Таким образом, необходимо показать, что если r — ранг $x.key$ в поддереве с корнем y в начале выполнения тела цикла, то в конце r становится рангом $x.key$ в поддереве, корнем которого является $y.p$. В каждой итерации цикла **while** мы рассматриваем поддерево, корнем которого является $y.p$. Мы уже подсчитали количество узлов в поддереве с корнем в узле y, который предшествует x при центрированном обходе дерева, так что теперь мы должны добавить узлы из поддерева, корнем которого является "брат" y (и который также предшествует x при центрированном обходе дерева), и добавить 1 для самого́ $y.p$, если, конечно, этот узел также предшествует x. Если y — левый дочерний узел, то ни $y.p$, ни любой узел из правого поддерева $y.p$ не может предшествовать x, так что r остается неизменным. В противном случае y является правым дочерним узлом, и все узлы в поддереве левого потомка $y.p$ предшествуют x, так же как и самому́ $y.p$. Соответственно, в строке 5 мы добавляем $y.p.left.size + 1$ к текущему значению r.

Завершение. Цикл завершается, когда $y = T.root$, так что поддерево, корнем которого является y, представляет собой все дерево целиком, и, таким образом, r является рангом $x.key$ в дереве в целом.

В качестве примера рассмотрим работу процедуры OS-RANK с деревом порядковой статистики, показанным на рис. 14.1. Если мы будем искать ранг узла с ключом 38, то получим следующую последовательность значений $y.key$ и r в начале цикла **while**.

Итерация	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

Процедура возвращает ранг 17.

Поскольку каждая итерация цикла **while** занимает время $O(1)$, а y при каждой итерации поднимается на один уровень вверх, общее время работы процедуры OS-RANK в наихудшем случае пропорционально высоте дерева, т.е. равно $O(\lg n)$ в случае дерева порядковой статистики с n узлами.

Поддержка размера поддеревьев

При наличии атрибута *size* в каждом узле процедуры OS-SELECT и OS-RANK позволяют быстро вычислять порядковые статистики. Однако этот атрибут будет совершенно бесполезным без корректного обновления базовыми модифицирующими операциями над красно-черными деревьями. Давайте рассмотрим, какие изменения нужно внести в алгоритмы вставки и удаления для того, чтобы они поддерживали поля размеров поддеревьев при сохранении асимптотического времени работы каждой операции.

В разделе 13.3 мы видели, что вставка в красно-черное дерево состоит из двух фаз. Первая фаза заключается в проходе вниз по дереву и вставке нового узла в качестве дочернего для уже существующего. Во второй фазе выполняется проход вверх по дереву, при котором выполняются изменения цветов узлов и повороты для сохранения красно-черных свойств дерева.

Для поддержки размеров поддеревьев в первой фазе достаточно просто увеличить значение $x.size$ для каждого узла x на простом пути от корня к листьям. Новый узел получает значение атрибута *size*, равное 1. Поскольку вдоль пути имеется $O(\lg n)$ узлов, дополнительное время, требующееся для поддержания атрибута *size* в первой фазе, составляет $O(\lg n)$.

Во второй фазе структурные изменения дерева вызываются только поворотами, которых, как мы знаем, может быть не больше двух. Кроме того, поворот является локальной операцией — после его выполнения становятся некорректными значения *size* только у двух узлов, вокруг связи между которыми выполняется поворот. Возвращаясь к коду LEFT-ROTATE(T, x) в разделе 13.2, необходимо просто добавить в него следующие строки.

```
y.size = x.size
x.size = x.left.size + x.right.size + 1
```

На рис. 14.2 проиллюстрировано обновление атрибутов. Изменения процедуры RIGHT-ROTATE симметричны только что рассмотренным.

Поскольку при вставке в красно-черное дерево выполняется не более двух поворотов, дополнительное время, требующееся для поддержки актуальности атрибутов *size* во второй фазе, равно $O(1)$. Таким образом, общее время встав-

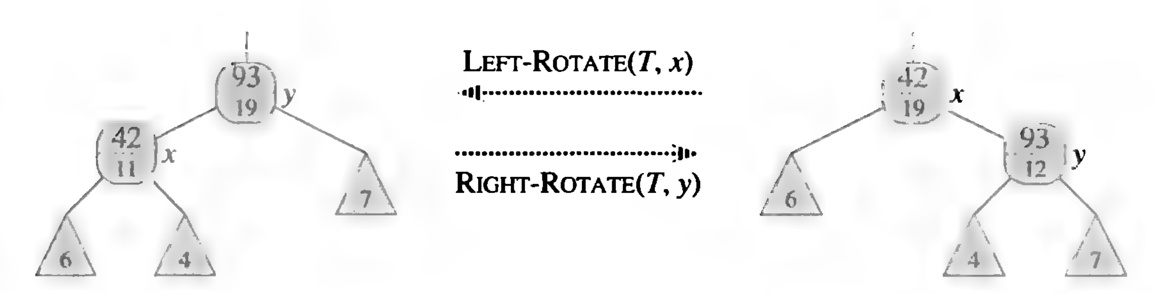


Рис. 14.2. Обновление размеров поддеревьев в процессе поворотов. Связь, вокруг которой осуществляется поворот, соединяет два узла, атрибуты *size* которых следует обновить. Обновления локальны, требуют информации из атрибутов *size* узлов x, y и корней поддеревьев, показанных в виде треугольников.

ки в дерево порядковой статистики с n узлами составляет $O(\lg n)$, т.е. остается асимптотически тем же, что и в случае обычного красно-черного дерева.

Удаление узла из красно-черного дерева также представляет собой двухфазный процесс — первая фаза удаляет узел из дерева поиска, лежащего в основе красно-черного дерева, а вторая восстанавливает красно-черные свойства, выполняя не более трех поворотов и не внося никаких других структурных изменений (см. раздел 13.4). В первой фазе из дерева извлекается узел y или выполняется его перемещение вверх по дереву. Для обновления размеров поддеревьев мы просто проходим по простому пути от узла y (начиная с его исходного положения в дереве) до корня дерева, уменьшая величину атрибута *size* каждого узла на нашем пути. Поскольку длина этого пути в красно-черном дереве с n узлами равна $O(\lg n)$, дополнительное время, затрачиваемое на поддержку атрибута *size* в первой фазе, составляет $O(\lg n)$. Во второй фазе обрабатываются $O(1)$ поворотов — тем же способом, что и в случае вставки. Итак, и вставка, и удаление в состоянии поддерживать корректность значений атрибутов *size* в дереве, при этом время их работы в дереве порядковой статистики с n узлами составляет $O(\lg n)$.

Упражнения

14.1.1

Покажите, как работает вызов процедуры OS-SELECT($T.root$, 10) для красно-черного дерева T, изображенного на рис. 14.1.

14.1.2

Покажите, как работает вызов процедуры OS-RANK(T, x) для дерева T, изображенного на рис. 14.1, если $x.key = 35$.

14.1.3

Разработайте нерекурсивную версию процедуры OS-SELECT.

14.1.4

Разработайте рекурсивную процедуру OS-KEY-RANK(T, k), которая получает в качестве входных параметров дерево порядковой статистики T и ключ k и возвращает значение ранга ключа k в динамическом множестве, представленном T. Считаем, что все ключи в T различны.

14.1.5

Даны элемент x дерева порядковой статистики с n узлами и неотрицательное целое число i. Каким образом можно найти i-й в порядке возрастания элемент, начиная отсчет от x, за время $O(\lg n)$?

14.1.6

Заметим, что процедуры OS-SELECT и OS-RANK используют атрибут *size* только для вычисления ранга узла. Предположим, что вместо этого в каждом узле хранится его ранг в поддереве, корнем которого он является. Покажите, каким образом можно поддерживать актуальность этой информации в процессе вставки и удаления (вспомните, что эти две операции могут выполнять повороты).

14.1.7

Покажите, как использовать дерево порядковой статистики для подсчета числа инверсий (см. задачу 2.4) в массиве размером n за время $O(n \lg n)$.

14.1.8 ⋆

Рассмотрим n хорд окружности, каждая из которых определяется своими конечными точками. Опишите алгоритм определения количества пар пересекающихся хорд за время $O(n \lg n)$. (Например, если все n хорд представляют собой диаметры, пересекающиеся в центре круга, то правильным ответом будет $\binom{n}{2}$). Считаем, что все конечные точки хорд различны.

14.2. Расширение структур данных

При разработке алгоритмов процесс расширения базовых структур данных для поддержки дополнительной функциональности встречается достаточно часто. В следующем разделе он будет использован для построения структур данных, которые поддерживают операции с промежутками. В данном разделе мы рассмотрим шаги, которые необходимо выполнить в процессе такого расширения, а также докажем теорему, которая во многих случаях позволяет упростить расширение красно-черных деревьев.

Расширение структур данных можно разбить на четыре шага.

1. Выбор базовой структуры данных.
2. Определение необходимой дополнительной информации, которую следует хранить в базовой структуре данных и актуальность которой следует поддерживать.
3. Проверка того, что дополнительная информация может поддерживаться основными модифицирующими операциями над базовой структурой данных.
4. Разработка новых операций.

Приведенные правила представляют собой общую схему, которой вы не обязаны жестко следовать. Проектирование — это искусство, зачастую опирающееся на

метод проб и ошибок, и все шаги могут на практике выполняться параллельно. Так, нет особого смысла в определении дополнительной информации и разработке новых операций (шаги 2 и 4), если мы не в состоянии эффективно поддерживать эту дополнительную информацию. Тем не менее описанная схема позволяет с пониманием дела направлять свои усилия, а также помочь в организации документирования расширенной структуры данных.

Мы следовали описанной схеме при разработке деревьев порядковой статистики в разделе 14.1. На шаге 1 в качестве базовой структуры данных мы выбрали красно-черные деревья в связи с их эффективной поддержкой других порядковых операций над динамическим множеством, таких как MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR.

На шаге 2 мы добавили в структуру данных атрибут *size*, в котором каждый узел x хранит размер своего поддерева. В общем случае дополнительная информация делает операции более эффективными, что наглядно видно на примере операций OS-SELECT и OS-RANK, которые можно было бы реализовать и с использованием одних лишь хранящихся в узлах ключей, но тогда они не могли бы выполняться за время $O(\lg n)$. Зачастую дополнительная информация представляет собой не данные, а указатели, как, например, в упр. 14.2.1.

На шаге 3 мы убедились в том, что модифицирующие операции вставки и удаления в состоянии поддерживать атрибут *size* с неизменным асимптотическим временем работы $O(\lg n)$. В идеале для поддержки дополнительной информации требуется обновлять только малую часть элементов структуры данных. Например, если хранить в каждом узле его ранг в дереве, то процедуры OS-SELECT и OS-RANK будут работать быстро, но вставка нового минимального элемента потребует при такой схеме внесения изменений в каждый узел дерева. При хранении размеров поддеревьев вставка нового элемента требует изменения информации только в $O(\lg n)$ узлах.

Шаг 4 состоял в разработке операций OS-SELECT и OS-RANK. В конце концов, именно необходимость новых операций, в первую очередь, приводит нас к расширению структуры данных. Иногда вместо разработки новых операций мы используем дополнительную информацию для ускорения существующих, как в упр. 14.2.1.

Расширение красно-черных деревьев

При использовании в качестве базовой структуры данных красно-черных деревьев можно доказать, что определенные виды дополнительной информации могут эффективно обновляться при вставках и удалениях, делая тем самым шаг 3 очень простым. Доказательство следующей теоремы аналогично рассуждениям из раздела 14.1 о возможности поддержки атрибута *size* деревьями порядковой статистики.

Теорема 14.1 (Расширение красно-черных деревьев)

Пусть f представляет собой атрибут, который расширяет красно-черное дерево T из n узлов, и пусть значение f каждого узла x зависит только от информации, хранящейся в узлах x, $x.left$ и $x.right$, возможно, включая $x.left.f$ и $x.right.f$.

В таком случае мы можем поддерживать актуальность информации f во всех узлах дерева T в процессе вставки и удаления без влияния на асимптотическое время работы данных процедур $O(\lg n)$.

Доказательство. Основная идея доказательства заключается в том, что изменение атрибута f узла x воздействует на значения атрибута f только у предков узла x. Иначе говоря, изменение $x.f$ может потребовать обновления $x.p.f$, но не более того; обновление $x.p.f$ может привести только к необходимости обновления $x.p.p.f$, и так далее вверх по дереву. При обновлении $T.root.f$ от этого значения не зависят никакие другие, так что процесс обновлений на этом завершается. Поскольку высота красно-черного дерева равна $O(\lg n)$, изменение атрибута f в некотором узле требует времени $O(\lg n)$ для обновления всех зависящих от него узлов.

Вставка узла x в дерево T состоит из двух фаз (см. раздел 13.3). Во время первой фазы узел x вставляется в дерево в качестве дочернего узла некоторого существующего узла $x.p$. Значение $x.f$ можно вычислить за время $O(1)$, поскольку в соответствии с условием теоремы оно зависит только от информации в других атрибутах x и информации в дочерних по отношению к x узлах; однако дочерними узлами x являются ограничители $T.nil$. После вычисления $x.f$ изменения распространяются вверх по дереву. Таким образом, общее время выполнения первой фазы вставки равно $O(\lg n)$. Во время второй фазы единственным преобразованием, способным вызвать структурные изменения дерева, являются повороты. Поскольку при повороте изменения затрагивают только два узла, общее время, необходимое для обновления атрибутов f, — $O(\lg n)$ на один поворот. Поскольку при вставке выполняется не более двух поворотов, общее время работы процедуры вставки равно $O(\lg n)$.

Так же, как и вставка, удаление выполняется в две стадии (см. раздел 13.4). Сначала выполняются изменения в дереве, при которых удаляемый узел извлекается из дерева. Если удаляемый узел имел в этот момент два дочерних узла, то следующий за ним в дереве узел перемещается в позицию удаленного узла. Распространение обновлений f имеют стоимость не более $O(\lg n)$ в силу локального характера вносимых изменений. На втором этапе при восстановлении красно-черных свойств выполняется не более трех поворотов, каждый из которых требует для распространения обновлений f времени, не превышающего $O(\lg n)$. Таким образом, общее время удаления составляет $O(\lg n)$. ■

Во многих случаях (в частности, в случае атрибутов *size* в деревьях порядковой статистики) время, необходимое для обновления атрибутов после вращения, составляет $O(1)$, а не $O(\lg n)$, полученное в доказательстве теоремы 14.1. Пример такого поведения можно найти в упр. 14.2.3.

Упражнения

14.2.1

Покажите, каким образом расширить дерево порядковой статистики, чтобы операции MAXIMUM, MINIMUM, SUCCESSOR и PREDECESSOR выполнялись за время

$O(1)$ в наихудшем случае. Асимптотическая производительность остальных операций над деревом порядковой статистики должна при этом остаться неизменной.

14.2.2

Может ли черная высота узлов в красно-черном дереве поддерживаться как атрибут узла дерева без изменения асимптотической производительности операций над красно-черными деревьями? Покажите, как этого достичь (или докажите, что это невозможно).

14.2.3 ★

Пусть $\otimes$ представляет собой ассоциативный бинарный оператор, а a — атрибут, поддерживаемый в каждом узле красно-черного дерева. Предположим, что мы хотим включить в каждый узел x дополнительный атрибут f, такой, что $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, где $x_1, x_2, \ldots, x_m$ — центрированный список узлов поддерева, корнем которого является x. Покажите, как обновлять атрибуты f после поворотов за время $O(1)$. Примените свои рассуждения, слегка их модифицировав, к атрибутам *size* в дереве порядковой статистики.

14.2.4 ★

Мы хотим добавить к красно-черным деревьям операцию RB-ENUMERATE(x, a, b), которая выводит все ключи $a \leq k \leq b$ в красно-черном дереве, корнем которого является x. Опишите, как можно реализовать процедуру RB-ENUMERATE, чтобы время ее работы составляло $\Theta(m + \lg n)$, где m — число выводимых ключей, а n — количество внутренних узлов в дереве. (*Указание:* нет необходимости добавлять новые атрибуты в красно-черное дерево.)

14.3. Деревья отрезков

В этом разделе мы расширим красно-черные деревья для поддержки операций над динамическими множествами отрезков. ***Отрезком*** называется упорядоченная пара действительных чисел $[t_1, t_2]$, таких, что $t_1 \leq t_2$. Отрезок $[t_1, t_2]$ представляет множество $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$. ***Интервал*** (t_1, t_2) представляет собой отрезок без конечных точек, т.е. множество $\{t \in \mathbb{R} : t_1 < t < t_2\}$, а ***полуинтервалы*** $[t_1, t_2)$ и $(t_1, t_2]$ образуются из отрезка при удалении из него одной из конечных точек. В случае, когда принадлежность концов несущественна, обычно говорят о ***промежутках***. В данном разделе мы будем работать с отрезками, но расширение результатов на интервалы и полуинтервалы не должно составить для читателя никакого труда.

Отрезки удобны для представления событий, которые занимают некоторый промежуток времени. Мы можем, например, сделать запрос к базе данных о том, какие события происходили в некоторый промежуток времени. Рассматриваемая в данном разделе структура данных обеспечивает эффективное средство для поддержки такой базы данных, работающей с промежутками.

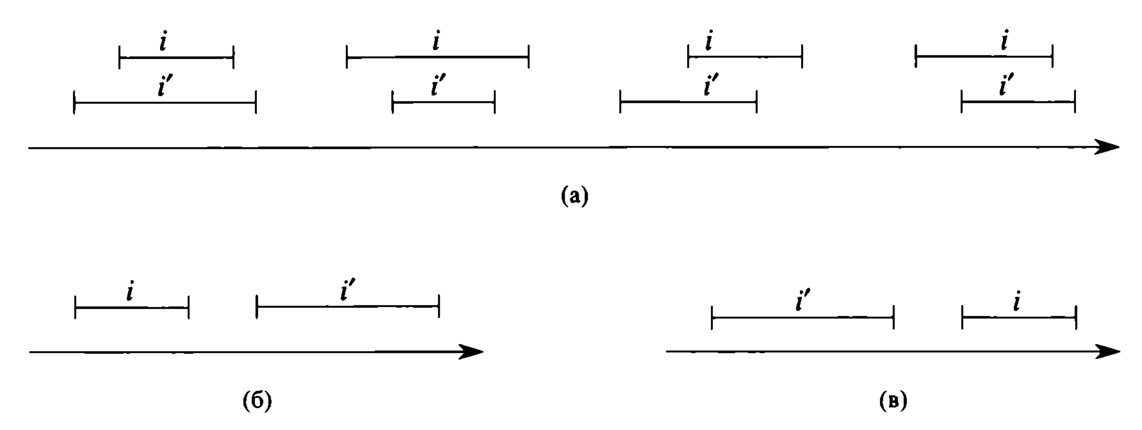


Рис. 14.3. Трихотомия отрезков i и i'. **(а)** Если i и i' перекрываются, возможны четыре ситуации; в каждой из них $i.low \le i'.high$ и $i'.low \le i.high$. **(б)** Отрезки не перекрываются и $i.high < i'.low$. **(в)** Отрезки не перекрываются и $i'.high < i.low$.

Мы можем представить отрезок $[t_1, t_2]$ в виде объекта i с атрибутами $i.low = t_1$ (***левый, или нижний, конец отрезка***) и $i.high = t_2$ (***правый, или верхний, конец***). Мы говорим, что отрезки i и i' ***перекрываются*** (overlap), если $i \cap i' \ne \emptyset$, т.е. если $i.low \le i'.high$ и $i'.low \le i.high$. Для любых двух отрезков i и i' выполняется только одно из трех свойств (трихотомия отрезков) (рис. 14.3):

а. i и i' перекрываются;

б. i находится слева от i' (т.е. $i.high < i'.low$);

в. i находится справа от i' (т.е. $i'.high < i.low$).

Дерево отрезков представляет собой красно-черное дерево, каждый элемент x которого содержит отрезок $x.int$. Деревья отрезков поддерживают следующие операции.

INTERVAL-INSERT(T, x) добавляет элемент x, атрибут int которого рассматривается как содержащий отрезок, в дерево отрезков T.

INTERVAL-DELETE(T, x) удаляет элемент x из дерева отрезков T.

INTERVAL-SEARCH(T, i) возвращает указатель на элемент x в дереве отрезков T, такой, что $x.int$ перекрывается с отрезком i, или указатель на ограничитель $T.nil$, если такого элемента во множестве нет.

На рис. 14.4 показано, как дерево отрезков представляет множество отрезков. Давайте рассмотрим этапы расширения структур данных из раздела 14.2 при решении задачи разработки дерева отрезков и реализации операций над ним.

Шаг 1. Базовая структура данных

В качестве базовой структуры данных мы выбираем красно-черное дерево, каждый узел x которого содержит отрезок $x.int$, а ключом узла является левый конец отрезка $x.int.low$. Таким образом, центрированный обход дерева приводит к перечислению отрезков в порядке сортировки по их левым концам.

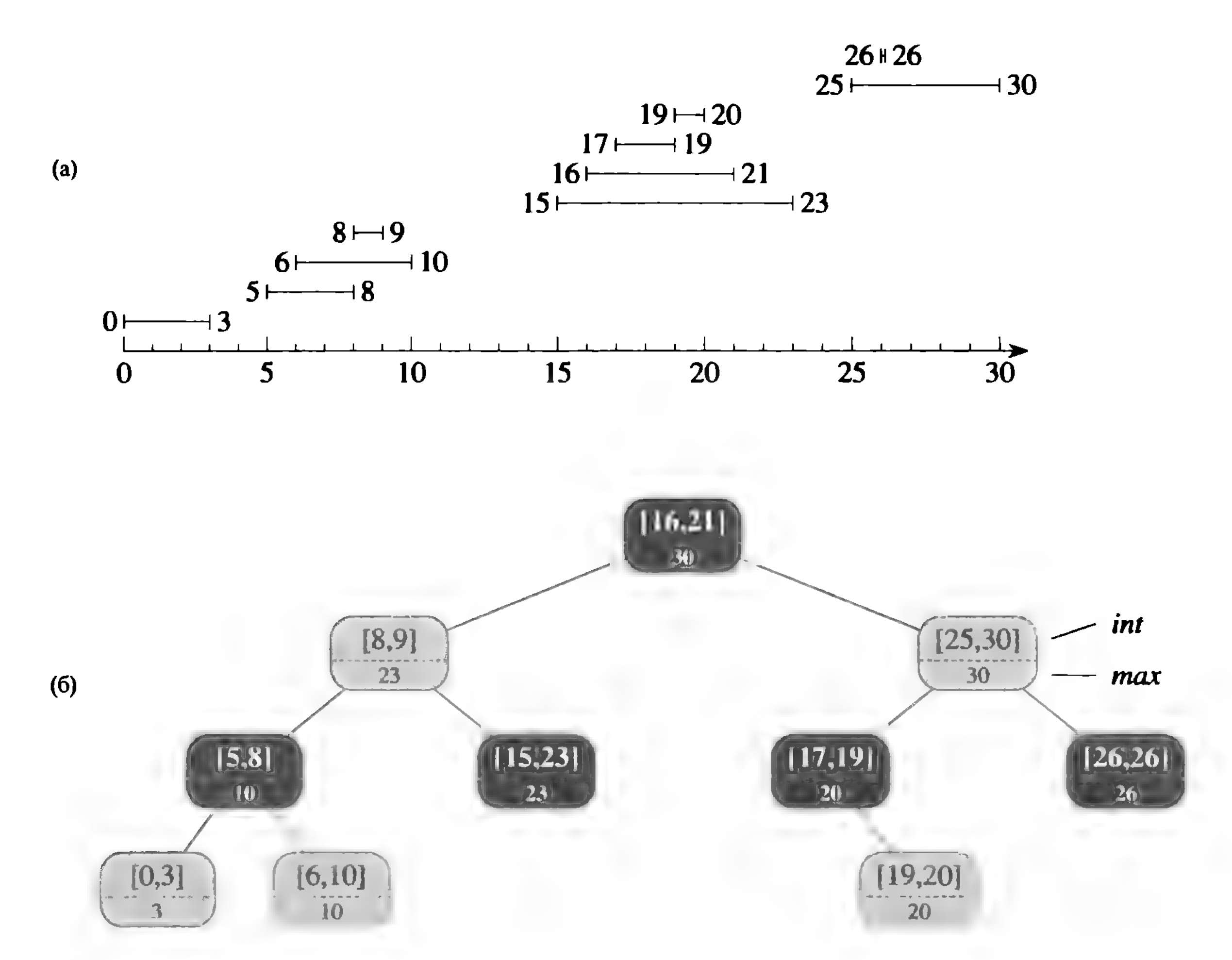


Рис. 14.4. Дерево отрезков. **(а)** Множество из десяти отрезков, отсортированных снизу вверх по левой конечной точке. **(б)** Дерево отрезков, представляющее это множество. Каждый узел x содержит отрезок, показанный над пунктирной чертой, и максимальное значение конечной точки для всех отрезков поддерева с корнем x, показанное под пунктирной линией. Центрированный обход дерева перечисляет все узлы в отсортированном по левым концам порядке.

Шаг 2. Дополнительная информация

В дополнение к самим отрезкам каждый узел x содержит значение $x.max$, которое представляет собой максимальное значение всех конечных точек отрезков, хранящихся в поддереве, корнем которого является x.

Шаг 3. Поддержка информации

Необходимо убедиться в том, что вставка и удаление в дереве с n узлами могут быть выполнены за время $O(\lg n)$. Определить значение атрибута max в узле x можно очень просто с использованием атрибутов max дочерних узлов:

$$x.max = \max(x.int.high, x.left.max, x.right.max) .$$

Таким образом, по теореме 14.1 вставка в дерево отрезков и удаление из него может быть выполнена за время $O(\lg n)$. В действительности, как показано в упр. 14.2.3 и 14.3.1, обновление атрибута max после вращения может быть выполнено за время $O(1)$.

Шаг 4. Разработка новых операций

Единственная новая операция, которую мы хотим разработать, — это INTERVAL-SEARCH(T, i), которая осуществляет поиск в дереве T отрезка, который перекрывается с данным. Если такого отрезка в дереве нет, процедура возвращает указатель на ограничитель $T.nil$.

```
INTERVAL-SEARCH(T, i)
x = T.root
while x ≠ T.nil и i не перекрывается с x.int
    if x.left ≠ T.nil и x.left.max ≥ i.low
        x = x.left
    else x = x.right
return x
```

Для поиска отрезка, который перекрывается с i, мы начинаем с присвоения указателю x корня дерева и выполняем спуск по дереву. Спуск завершается, когда мы находим перекрывающийся отрезок или когда x указывает на ограничитель $T.nil$. Поскольку каждая итерация основного цикла выполняется за время $O(1)$, а высота красно-черного дерева с n узлами равна $O(\lg n)$, время работы процедуры INTERVAL-SEARCH равно $O(\lg n)$.

Прежде чем убедиться в корректности процедуры INTERVAL-SEARCH, давайте посмотрим, как она работает, на примере дерева, показанного на рис. 14.4. Предположим, что мы хотим найти отрезок, перекрывающийся с отрезком $i = [22, 25]$. Мы начинаем работу с корня, в котором содержится отрезок $[16, 21]$, который не перекрывается с отрезком i. Поскольку значение $x.left.max = 23$ превышает $i.low = 22$, цикл продолжает выполнение с x, указывающим на левый дочерний узел корня. Этот узел содержит отрезок $[8, 9]$, который также не перекрывается с отрезком i. Теперь $x.left.max = 10$ меньше $i.low = 22$, так что мы переходим к правому дочернему узлу. В нем содержится отрезок $[15, 23]$, перекрывающийся с x, так что процедура возвращает указатель на данный узел.

В качестве примера неудачного поиска попробуем найти в том же дереве отрезок, перекрывающийся с отрезком $i = [11, 14]$. Мы вновь начинаем с корня. Поскольку отрезок в корне $[16, 21]$ не перекрывается с i и поскольку $x.left.max = 23$ больше, чем $i.low = 11$, мы переходим к левому дочернему узлу корня с отрезком $[8, 9]$. Отрезок $[8, 9]$ также не перекрывается с i, а $x.left.max = 10$, что меньше, чем $i.low = 11$, поэтому мы переходим вправо (обратите внимание, что теперь в левом поддереве нет ни одного отрезка, перекрывающегося с i). Отрезок $[15, 23]$ не перекрывается с i, его левый дочерний узел — $T.nil$, так что цикл завершается и процедура возвращает указатель на ограничитель $T.nil$.

Для того чтобы убедиться в корректности процедуры INTERVAL-SEARCH, нужно разобраться, почему для поиска достаточно пройти по дереву всего лишь по одному пути от корня. Основная идея заключается в том, что в любом узле x, если $x.int$ не перекрывается с i, дальнейший поиск всегда идет в безопасном направлении, т.е. перекрывающийся отрезок, если таковой имеется в дереве, гарантированно будет обнаружен в исследуемой части дерева. Более точно это свойство сформулировано в следующей теореме.

Теорема 14.2

Любой вызов процедуры INTERVAL-SEARCH(T, i) возвращает либо узел, отрезок которого перекрывается с отрезком i, либо, если в дереве T не содержится отрезка, перекрывающегося с i, $T.nil$.

Доказательство. Цикл **while** в строках 2–5 завершается, если $x = T.nil$ либо если i перекрывается с $x.int$. В последнем случае процедура тривиально возвращает корректное значение x. Поэтому нас интересует первый случай, когда цикл завершается из-за того, что $x = T.nil$.

Воспользуемся следующим инвариантом цикла **while** в строках 2–5.

Если дерево T содержит отрезок, который перекрывается с i, то этот отрезок находится в поддереве, корнем которого является узел x.

Исследуем этот инвариант цикла как обычно.

Инициализация. Перед выполнением первой итерации в строке 1 переменной x присваивается указатель на корень дерева T, так что инвариант выполняется.

Сохранение. При каждой итерации цикла **while** выполняется либо строка 4, либо строка 5. Покажем, что инвариант цикла сохраняется в любом случае.

Если выполняется строка 5, то в силу ветвления в строке 3 мы имеем $x.left = T.nil$ или $x.left.max < i.low$. Если $x.left = T.nil$, поддерево с корнем $x.left$, очевидно, не содержит отрезка, перекрывающегося с i, и присвоение x значения $x.right$ сохраняет инвариант. Предположим, следовательно, что $x.left \neq T.nil$ и $x.left.max < i.low$. Как показано на рис. 14.5, (а), для каждого отрезка i' в левом поддереве x имеем

$$\begin{aligned} i'.high &\leq x.left.max \\ &< i.low \ . \end{aligned}$$

Согласно трихотомии отрезков i' и i не перекрываются. Таким образом, левое поддерево x не содержит отрезков, перекрывающихся с i, так что присвоение x значения $x.right$ сохраняет инвариант.

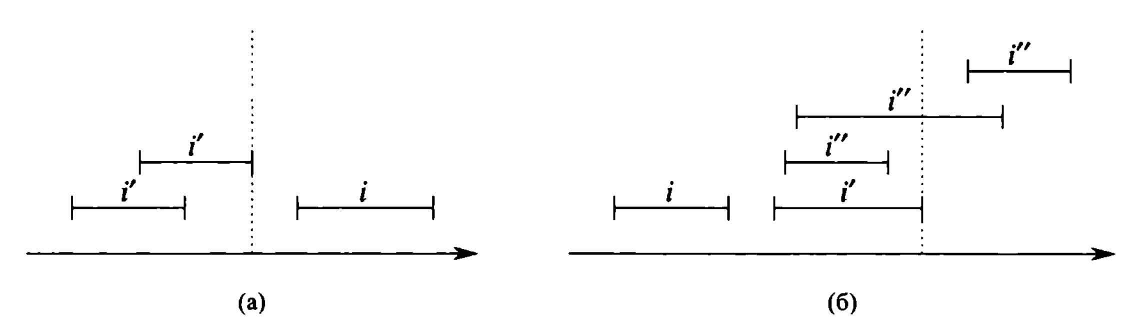


Рис. 14.5. Отрезки в доказательстве теоремы 14.2. Значение $x.left.max$ показано в обоих случаях пунктирной линией. **(а)** Поиск выполняется вправо. В левом поддереве x нет отрезка i', перекрывающегося с i. **(б)** Поиск выполняется влево. Левое поддерево x содержит отрезок, перекрывающийся с i (ситуация не показана), либо отрезок i', такой, что $i'.high = x.left.max$. Поскольку i не перекрывается с i', он не перекрывается и ни с одним из отрезков i'' в правом поддереве x, поскольку $i'.low \leq i''.low$.

Если же выполняется строка 4, то, как мы покажем, если в левом поддереве x нет отрезка, перекрывающегося с i, то его вообще нет в дереве. Поскольку выполняется строка 4, в силу условия в строке 3 имеем $x.left.max \geq i.low$. Кроме того, по определению атрибута max в левом поддереве x должен быть некоторый интервал i', такой, что

$$\begin{aligned} i'.high &= x.left.max \\ &\geq i.low \ . \end{aligned}$$

(Эта ситуация проиллюстрирована на рис. 14.5, (б).) Поскольку i и i' не перекрываются и поскольку неверно, что $i'.high < i.low$, отсюда в соответствии со свойством трихотомии отрезков следует, что $i.high < i'.low$. Поскольку дерево отрезков упорядочено в соответствии с левыми концами отрезков, из свойства дерева поиска вытекает, что для любого отрезка i'' из правого поддерева x

$$\begin{aligned} i.high &< i'.low \\ &\leq i''.low \ . \end{aligned}$$

Из трихотомии отрезков следует, что i и i'' не перекрываются. Мы можем, таким образом, заключить, что независимо от того, имеется ли в левом поддереве x отрезок, перекрывающийся с i, присвоение x значения $x.left$ сохраняет инвариант цикла.

Завершение. Если цикл завершается по условию $x = T.nil$, то в дереве, корнем которого является x, нет отрезков, перекрывающихся с i. Обращение инварианта цикла приводит к заключению, что дерево T не содержит отрезков, перекрывающихся с i. Следовательно, возвращаемое процедурой значение $T.nil$ совершенно корректно. ■

Таким образом, процедура INTERVAL-SEARCH работает корректно.

Упражнения

14.3.1

Напишите псевдокод процедуры LEFT-ROTATE, которая работает с узлами дерева отрезков и обновляет атрибуты max за время $O(1)$.

14.3.2

Перепишите код процедуры INTERVAL-SEARCH для корректной работы с интервалами (отрезками без конечных точек).

14.3.3

Разработайте эффективно работающий алгоритм, который для данного отрезка i возвращает отрезок, перекрывающийся с i и имеющий минимальное значение левого конца (либо $T.nil$, если такого отрезка не существует).

14.3.4

Пусть имеется дерево отрезков T и отрезок i. Опишите, каким образом найти в дереве T все отрезки, перекрывающиеся с отрезком i, за время $O(\min(n, k \lg n))$, где k — количество отрезков в выводимом списке. (*Указание:* один простой метод выполняет запросы, меняя между ними само дерево. Попробуйте найти немного более сложное решение, не изменяющее дерево.)

14.3.5

Какие изменения следует внести в процедуры дерева отрезков для поддержки новой операции INTERVAL-SEARCH-EXACTLY(T, i), которая получает в качестве параметров дерево отрезков T и отрезок i и возвращает указатель на узел x, такой, что $x.int.low = i.low$ и $x.int.high = i.high$ (либо $T.nil$, если такого узла в дереве T нет). Все операции, включая INTERVAL-SEARCH-EXACTLY, должны выполняться в дереве с n узлами за время $O(\lg n)$.

14.3.6

Пусть есть динамическое множество чисел Q, поддерживающее операцию MIN-GAP, возвращающую минимальное расстояние между соседними числами в Q. Например, если $Q = \{1, 5, 9, 15, 18, 22\}$, то MIN-GAP$(Q)$ возвратит значение $18 - 15 = 3$, так как 15 и 18 — ближайшие соседние числа в Q. Разработайте максимально эффективные процедуры INSERT, DELETE, SEARCH и MIN-GAP и проанализируйте их время работы.

14.3.7 ★

Базы данных при разработке СБИС зачастую представляют интегральную схему как список прямоугольников. Предположим, что все прямоугольники ориентированы вдоль осей x и y, так что представление прямоугольника состоит из минимальных и максимальных координат x и y. Разработайте алгоритм для выяснения, имеются ли в данном множестве из n прямоугольников два перекрывающихся (искать все перекрывающиеся пары не нужно). Перекрытием считается также ситуация, когда один прямоугольник лежит полностью внутри другого, пусть при этом их границы и не пересекаются. Время работы алгоритма должно составлять $O(n \lg n)$. (*Указание:* перемещайте "строку развертки" по множеству прямоугольников.)

Задачи

14.1. Точка максимального перекрытия

Предположим, что необходимо найти ***точку максимального перекрытия*** множества отрезков, т.е. точку, в которой перекрывается наибольшее количество отрезков множества.

а. Покажите, что такая точка всегда имеется и представляет собой конечную точку одного из отрезков.

б. Разработайте структуру данных, которая поддерживает эффективную работу операций INTERVAL-INSERT, INTERVAL-DELETE и FIND-POM (которая возвращает точку максимального перекрытия). (*Указание:* воспользуйтесь красно-черным деревом всех конечных точек отрезков. С каждым левым концом отрезка связано значение $+1$, с правым — значение -1. Добавьте к узлам дерева некоторую дополнительную информацию для поддержки точки максимального перекрытия.)

14.2. Перестановка Иосифа

Задача Иосифа[1] формулируется следующим образом. Предположим, что n человек расставлены по кругу и задано некоторое натуральное число $m \leq n$. Начиная с определенного человека, мы идем по кругу, удаляя каждого m-го человека. После удаления человека счет продолжается дальше. Процесс продолжается, пока все n человек не будут удалены. Порядок, в котором люди удаляются из круга, определяет ***(n, m)-перестановку Иосифа*** целых чисел $1, 2, \ldots, n$. Например, $(7, 3)$-перестановка Иосифа имеет вид $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

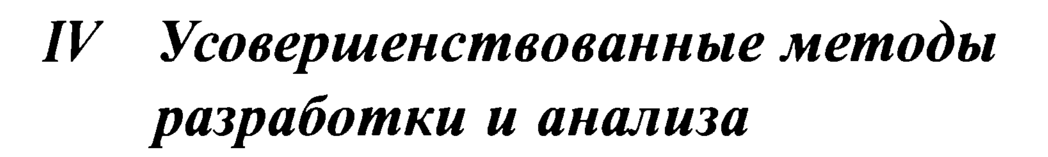
а. Пусть m — некоторая фиксированная константа. Разработайте алгоритм, который для данного n за время $O(n)$ выводит (n, m)-перестановку Иосифа.

б. Пусть m не является константой. Разработайте алгоритм, который для данных n и m за время $O(n \lg n)$ выводит (n, m)-перестановку Иосифа.

Заключительные замечания

В книге [280] Препарата (Preparata) и Шамос (Shamos) описывают ряд деревьев отрезков, встречавшихся литературе, и приводят результаты из работ Эдельсбруннера (H. Edelsbrunner, 1980) и Мак-Крейта (McCreight, 1981). В книге детально описано дерево отрезков, в котором для данной статической базы данных из n отрезков все k отрезков, перекрывающихся с заданным, могут быть найдены за время $O(k + \lg n)$.

[1]Достаточно подробно об этой задаче, ее происхождении и вариациях, можно прочесть, например, в книге У. Болл, Г. Коксетер. *Математические эссе и развлечения.* — М.: Мир, 1986. — С. 43–47. — *Примеч. ред.*



IV Усовершенствованные методы разработки и анализа

Введение

В этой части описаны три важных метода разработки и анализа эффективных алгоритмов: динамическое программирование (глава 15), жадные алгоритмы (глава 16) и амортизационный анализ (глава 17). В предыдущих частях были представлены другие широко распространенные методы, такие как метод “разделяй и властвуй”, рандомизация и решение рекуррентных соотношений. Новые подходы, с которыми вам предстоит ознакомиться в этой части, более сложные, однако они полезны для решения многих вычислительных задач. К темам, рассмотренным в данной части, мы еще обратимся позже в данной книге.

Динамическое программирование обычно находит применение в задачах оптимизации, в которых для получения оптимального решения необходимо сделать определенное множество выборов. После того как каждый из выборов сделан, часто возникают вспомогательные подзадачи того же вида. Динамическое программирование эффективно тогда, когда определенная вспомогательная подзадача может возникнуть в результате нескольких вариантов выбора. Основной метод решения таких задач заключается в сохранении решения каждой подзадачи, которая может возникнуть повторно. В главе 15 показано, как благодаря этой простой идее алгоритм, время решения которого экспоненциально зависит от объема входных данных, иногда можно преобразовать в алгоритм с полиномиальным временем работы.

Жадные алгоритмы, подобно алгоритмам, применяемым в динамическом программировании, используются в задачах оптимизации, для рационального решения которых требуется сделать ряд выборов. Идея, лежащая в основе жадного алгоритма, заключается в том, чтобы каждый выбор был локально оптимальным. В качестве простого примера приведем задачу о выдаче сдачи: чтобы свести к минимуму количество монет, необходимых для выдачи определенной суммы, достаточно каждый раз выбирать монету наибольшего достоинства, не превышающую

той суммы, которую осталось выдать.[1] Можно сформулировать много таких задач, оптимальное решение которых с помощью жадных алгоритмов получается намного быстрее, чем с помощью методов динамического программирования. Однако не всегда просто выяснить, окажется ли эффективным жадный алгоритм. В главе 16 приводится обзор теории матроида, которая часто оказывается полезной для принятия подобных решений.

Амортизационный анализ — это средство анализа алгоритмов, в которых выполняется последовательность однотипных операций. Вместо того чтобы накладывать границы на время выполнения каждой операции, с помощью амортизационного анализа оценивается длительность работы всей последовательности в целом. Одна из причин эффективности этой идеи заключается в том, что в некоторых последовательностях операций невозможна ситуация, когда время работы всех индивидуальных операций является наихудшим. Зачастую одни операции в таких последовательностях оказываются дорогостоящими в плане времени работы, в то время как многие другие — дешевыми. Заметим, что амортизационный анализ — это не просто средство анализа. Его можно рассматривать и как метод разработки алгоритмов, поскольку разработка и анализ времени работы алгоритмов часто тесно переплетаются. В главе 17 излагаются основы трех способов амортизационного анализа алгоритмов.

---

[1]Следует отметить, что возможны такие экзотические наборы монет, когда жадный алгоритм дает неверное решение. — *Примеч. ред.*

Глава 15. Динамическое программирование

Динамическое программирование, как и метод “разделяй и властвуй”, позволяет решать задачи, комбинируя решения вспомогательных подзадач. (Термин “программирование” в данном контексте означает табличный метод, а не составление компьютерного кода.) Мы уже видели в главах 2 и 4, как в алгоритмах “разделяй и властвуй” задача делится на несколько независимых подзадач, каждая из которых решается рекурсивно, после чего из решений подзадач формируется решение исходной задачи. Динамическое программирование, напротив, находит применение тогда, когда подзадачи перекрываются, т.е. когда разные подзадачи используют решения одних и тех же подзадач. В этом контексте алгоритм “разделяй и властвуй”, многократно решая задачи одних и тех же типов, выполняет больше действий, чем необходимо. В алгоритме динамического программирования каждая подзадача решается только один раз, после чего ответ сохраняется в таблице. Это позволяет избежать одних и тех же повторных вычислений каждый раз, когда встречается данная, уже решенная ранее, подзадача.

Динамическое программирование, как правило, применяется к ***задачам оптимизации*** (optimization problems). Такая задача может иметь много возможных решений. С каждым вариантом решения можно сопоставить какое-то значение, и нам нужно найти среди них решение с оптимальным (минимальным или максимальным) значением. Назовем такое решение *одним из возможных* оптимальных решений. В силу того, что таких решений с оптимальным значением может быть несколько, следует отличать их от *единственного* оптимального решения.[1]

Процесс разработки алгоритмов динамического программирования можно разбить на четыре перечисленных ниже этапа.

1. Описание структуры оптимального решения.
2. Определение значения, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

[1]В оригинале использованы различные артикли и говорится об “*an* optimal solution” и “*the* optimal solution”. — *Примеч. пер.*

Длина i	1	2	3	4	5	6	7	8	9	10
Цена p_i	1	5	8	9	10	17	17	20	24	30

Рис. 15.1. Пример таблицы цен отрезков стержня. Каждый отрезок длиной i приносит компании прибыль p_i.

Этапы 1–3 составляют основу метода динамического программирования для решения задач. Этап 4 может быть опущен, если требуется узнать только значение, соответствующее оптимальному решению. На этапе 4 иногда используется дополнительная информация, полученная на этапе 3, что облегчает процесс конструирования оптимального решения.

В последующих разделах метод динамического программирования используется для решения некоторых задач оптимизации. В разделе 15.1 исследуется задача разрезания стержня на меньшие стержни таким образом, чтобы получить за них максимальную прибыль. В разделе 15.2 исследуется вопрос о том, в каком порядке следует выполнять перемножение нескольких матриц, чтобы свести к минимуму общее количество операций перемножения скаляров. На этих двух примерах в разделе 15.3 рассматриваются две основные характеристики, которыми должны обладать задачи, для которых подходит метод динамического программирования. Далее, в разделе 15.4, показано, как найти самую длинную общую подпоследовательность двух последовательностей. Наконец в разделе 15.5 с помощью динамического программирования конструируется дерево бинарного поиска, оптимальное для заданного распределения ключей, по которым ведется поиск.

15.1. Разрезание стержня

В нашем первом примере динамическое программирование применяется для решения простой задачи о том, как лучше разрезать стальные стержни. Компания "Навар лимитед" покупает длинные стальные стержни, режет их на куски и продает. Сама порезка стержней не стоит компании ни копейки. Руководство "Навар лимитед" хочет знать, как лучше всего разрезать стержни на части.

Предположим, что нам известны цены p_i ($i = 1, 2, \ldots$), по которым компания продает куски длиной i. Длины кусков всегда представляют собой целые числа. На рис. 15.1 показан пример таблицы цен.

Задача разрезания стержня формулируется следующим образом. Имеются стержень длиной n и таблица цен p_i для $i = 1, 2, \ldots, n$. Необходимо найти максимальную прибыль r_n, получаемую при разрезании стержня и продаже полученных кусков. Заметим, что если цена p_n стержня длиной n достаточно велика, оптимальное решение может состоять в продаже стержня целиком, без разрезов.

Рассмотрим случай $n = 4$. На рис. 15.2 показаны все способы разрезания стержня длиной 4, включая вариант оставить его нетронутым. Как видно из рисунка, оптимальным является разрезание стержня длиной 4 на два куска длиной 2, что приводит к прибыли $p_2 + p_2 = 5 + 5 = 10$.

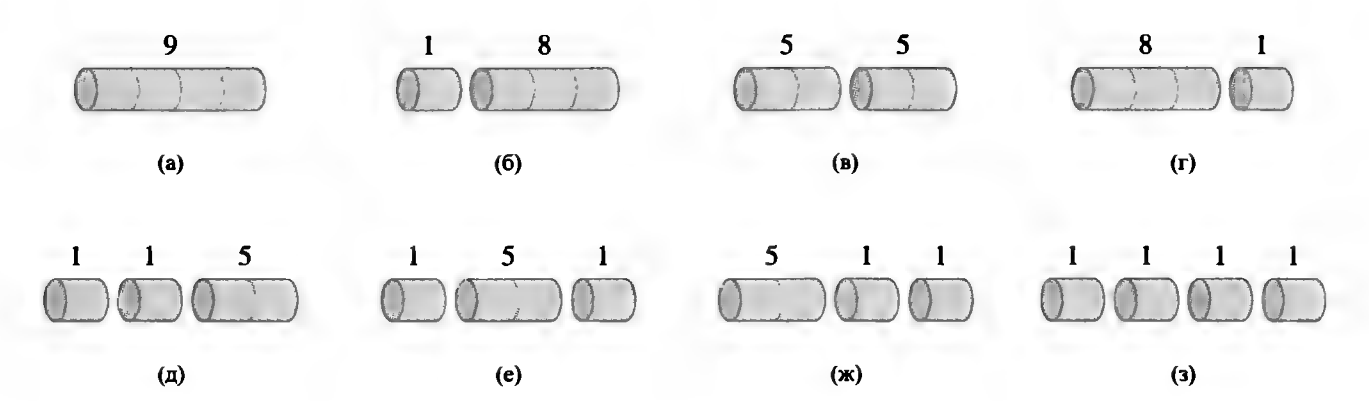


Рис. 15.2. Восемь возможных способов разрезания стержня длиной 4. Над каждым фрагментом стержня указана его цена в соответствии с таблицей на рис. 15.1. Оптимальная стратегия показана в части (в): она заключается в том, чтобы разрезать стержень на две части длиной 2 каждая с общей ценой 10.

Стержень длиной n можно разрезать 2^{n-1} разными способами, поскольку мы можем независимо выбирать, резать его или нет на расстоянии i от левого конца, где $i = 1, 2, \ldots, n-1$.[2] Мы будем записывать разбиение на части в виде обычного сложения, так что запись $7 = 2 + 2 + 3$ означает, что стержень длиной 7 разрезан на три части: две длиной 2 и одну длиной 3. Если оптимальное решение состоит в разрезании стержня на k частей, для некоторого $1 \le k \le n$, то оптимальное разбиение

$$n = i_1 + i_2 + \cdots + i_k$$

стержня на части длиной i_1, i_2, ..., i_k дает соответствующую максимальную прибыль

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} \ .$$

В случае нашей конкретной задачи максимальные прибыли r_i, $i = 1, 2, \ldots, 10$, получаются с помощью следующих разбиений:

$$\begin{aligned}
r_1 &= 1 \quad \text{из решения } 1 = 1 \quad \text{(без разрезов)} \ , \\
r_2 &= 5 \quad \text{из решения } 2 = 2 \quad \text{(без разрезов)} \ , \\
r_3 &= 8 \quad \text{из решения } 3 = 3 \quad \text{(без разрезов)} \ , \\
r_4 &= 10 \ \text{из решения } 4 = 2 + 2 \ , \\
r_5 &= 13 \ \text{из решения } 5 = 2 + 3 \ , \\
r_6 &= 17 \ \text{из решения } 6 = 6 \quad \text{(без разрезов)} \ , \\
r_7 &= 18 \ \text{из решения } 7 = 1 + 6 \ \text{ или } \ 7 = 2 + 2 + 3 \ , \\
r_8 &= 22 \ \text{из решения } 8 = 2 + 6 \ , \\
r_9 &= 25 \ \text{из решения } 9 = 3 + 6 \ , \\
r_{10} &= 30 \ \text{из решения } 10 = 10 \quad \text{(без разрезов)} \ .
\end{aligned}$$

---

[2]Если потребовать, чтобы отрезаемые части располагались в неубывающем порядке, то придется рассмотреть меньшее количество вариантов разрезания. При $n = 4$ следует рассмотреть только 5 таких способов: части (а), (б), (в), (д) и (з) на рис. 15.2. Количество способов разрезания именуется ***функцией разбиения*** (partition function); оно приблизительно равно $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. Эта величина меньше, чем 2^{n-1}, но все равно больше любого полинома от n. Однако мы не будем подробно исследовать этот вопрос.

В общем случае можно записать значения r_n для $n \geq 1$ через оптимальные прибыли от более коротких стержней:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \ . \tag{15.1}$$

Первый аргумент, p_n, соответствует продаже стержня длиной n как есть, без разрезов. Прочие $n - 1$ аргументов функции max соответствуют максимальным доходам, получаемым при первоначальном разрезании стержня на две части размерами i и $n - i$, для каждого $i = 1, 2, \ldots, n - 1$, с последующим оптимальным разрезанием второй части (при этом от полученных частей мы получаем доходы r_i и r_{n-i}). Поскольку заранее неизвестно, какое значение i оптимизирует прибыль, придется рассмотреть все возможные значения i и выбрать из них то, которое максимизирует доход. Кроме того, возможно, следует не выбирать ни одного значения i вообще, а предпочесть продавать стержни неразрезанными.

Заметим, что для решения исходной задачи размером n мы решаем меньшие задачи того же вида. Как только мы сделали первый разрез, мы можем рассматривать две части стержня как независимые экземпляры задачи разрезания стержня. Общее оптимальное решение включает оптимальные решения двух связанных подзадач, максимизирующих доходы от каждой из двух частей стержня. Мы говорим, что задача разрезания стержня демонстрирует ***оптимальную подструктуру***: оптимальное решение задачи включает оптимальные решения подзадач, которые могут быть решены независимо.

В связанном, но немного более простом, способе организации рекурсивной структуры задачи разрезания стержня мы рассматриваем разрезание стержня как состоящее из первой части длиной i и остатка длиной $n - i$. Первая часть далее не разрезается; резать можно только остаток. Таким образом, мы можем рассматривать любое разрезание стержня длиной n как первую часть и некоторое разделение на части остатка стержня без первой части. При этом допускается и решение без разрезов, если первая часть имеет размер $i = n$ и дает прибыль p_n, а остаток длиной 0 дает нулевой доход. В итоге мы получаем более простую версию уравнения (15.1):

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \ . \tag{15.2}$$

В такой формулировке оптимальное решение включает решение только *одной* связанной подзадачи — разрезания остатка — вместо двух, как это было ранее.

Рекурсивная нисходящая реализация

Приведенная далее процедура реализует вычисления, неявно заключенные в уравнении (15.2), простым нисходящим рекурсивным способом.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Процедура CUT-ROD получает в качестве входных данных массив цен $p[1..n]$ и целое число n и возвращает максимально возможную прибыль для стержня длиной n. Если $n = 0$, прибыль невозможна, так что процедура CUT-ROD возвращает 0 в строке 2. Строка 3 инициализирует максимальную прибыль q значением $-\infty$, так что цикл **for** в строках 4 и 5 корректно вычисляет $q = \max_{1 \leq i \leq n}(p_i + \text{CUT-ROD}(p, n-i))$; затем строка 6 возвращает вычисленное значение. Простая индукция по n с использованием уравнения (15.2) доказывает, что этот ответ равен искомому значению r_n.

Если вы закодируете процедуру CUT-ROD на своем любимом языке программирования, то обнаружите, что, когда входные размеры становятся умеренно большими, программа начинает работать весьма медленно. При $n = 40$ программа на вашем компьютере будет работать как минимум несколько минут, а скорее всего — больше часа. Вы можете заметить, что при увеличении n на 1 время работы вашей программы примерно удваивается.

Почему же процедура CUT-ROD столь неэффективна? Дело в том, что она рекурсивно вызывает сама себя вновь и вновь с одними и теми же значениями параметров; она многократно решает одни и те же подзадачи. На рис. 15.3 показано, что происходит при $n = 4$: CUT-ROD(p, n) вызывает CUT-ROD$(p, n-i)$ для $i = 1, 2, \ldots, n$; или, что то же самое, CUT-ROD(p, n) вызывает CUT-ROD(p, j) для каждого $j = 0, 1, \ldots, n-1$. Когда этот процесс рекурсивно разворачивается, рост количества выполняемой работы как функции от n носит взрывной характер.

Чтобы проанализировать время работы процедуры CUT-ROD, обозначим через $T(n)$ общее количество вызовов CUT-ROD со вторым параметром, равным n. Это выражение равно числу узлов в поддереве с корнем с меткой n в дереве рекурсии. Это число включает и начальный вызов в корне. Таким образом, $T(0) = 1$ и

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \ . \tag{15.3}$$

Начальная единица соответствует вызову в корне, а член $T(j)$ учитывает количество вызовов (включая рекурсивные), связанных с вызовом CUT-ROD$(p, n-i)$, где $j = n - i$. В упр. 15.1.1 требуется показать, что

$$T(n) = 2^n \ , \tag{15.4}$$

так что время работы процедуры CUT-ROD экспоненциально зависит от n.

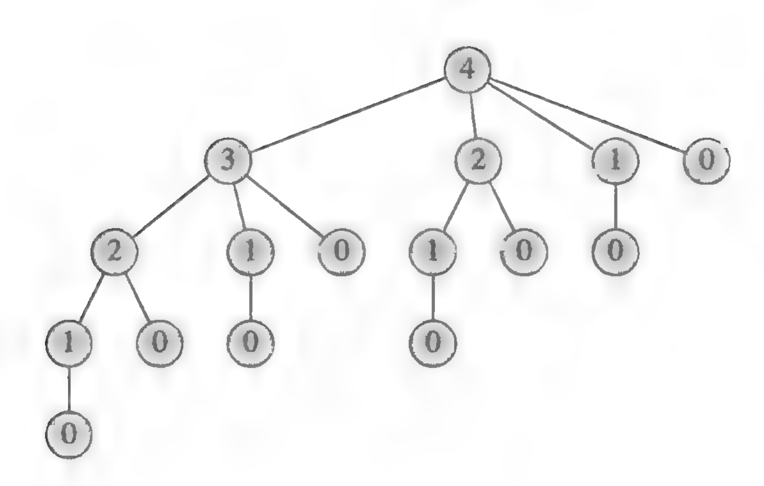


Рис. 15.3. Дерево рекурсии показывает рекурсивные вызовы, являющиеся результатом вызова CUT-ROD(p, n) для $n = 4$. Каждая метка узла указывает размер n соответствующей подзадачи, так что ребро от родительского узла с меткой s к дочернему узлу с меткой t соответствует отрезанию от стержня части размером $s - t$ и решение подзадачи для оставшейся части размером t. Путь от корня к листу соответствует одному из 2^{n-1} способов разрезания стержня длиной n. В общем случае это дерево рекурсии имеет 2^n узлов и 2^{n-1} листьев.

В ретроспективе в этом экспоненциальном времени нет ничего удивительного. Процедура CUT-ROD явным образом рассматривает все 2^{n-1} возможных способа разрезания стержня длиной n. Дерево рекурсивных вызовов имеет 2^{n-1} листьев, по одному для каждого возможного разрезания стержня. Метки на простом пути от корня к листу указывают размеры правой части стержня перед каждым разрезанием. Иначе говоря, метки указывают соответствующие точки разрезов, отмеренные от правого конца стержня.

Применение динамического программирования для оптимального разрезания стержня

Теперь мы покажем, как превратить процедуру CUT-ROD в эффективный алгоритм, воспользовавшись динамическим программированием.

Метод динамического программирования работает следующим образом. Заметив, что имеющееся рекурсивное решение неэффективно из-за того, что многократно решаются одни и те же подзадачи, мы будем сохранять их решения, тем самым добиваясь только *однократного* решения подзадач. Если позже нам вновь придется решать такую подзадачу, мы просто найдем ее ответ, не решая ее заново. Таким образом, динамическое программирование использует дополнительную память для экономии времени вычисления; это один из примеров ***пространственно-временного компромисса***. Экономия времени работы может быть очень большой: решение с экспоненциальным временем работы можно превратить в решение с полиномиальным временем. Подход с применением динамического программирования решает поставленную задачу за полиномиальное время, если количество *различных* подзадач полиномиально зависит от размера входных данных, и мы можем решить каждую из них за полиномиальное время.

Обычно имеется два эквивалентных способа реализации подхода динамического программирования. Мы проиллюстрируем оба на примере задачи о разрезании стержня.

Первый подход — ***нисходящий с запоминанием*** (memoization[3]). При таком подходе мы пишем процедуру рекурсивно, как обычно, но модифицируем ее таким образом, чтобы она запоминала решение каждой подзадачи (обычно в массиве или хеш-таблице). Теперь процедура первым делом проверяет, не была ли эта задача решена ранее. Если была, то возвращается сохраненное значение (и экономятся вычисления на данном уровне). Если же подзадача еще не решалась, процедура вычисляет возвращаемое значение, как обычно. Мы говорим, что данная рекурсивная процедура ***с запоминанием*** — она "запоминает" вычисленный ею результат.

Второй подход — ***восходящий***. Обычно он зависит от некоторого естественного понятия "размера" подзадачи, такого, что решение любой конкретной подзадачи зависит только от решения "меньших" подзадач. Мы сортируем подзадачи по размерам в возрастающем порядке. При решении определенной подзадачи необходимо решить все меньшие подзадачи, от которых она зависит, и сохранить полученные решения. Каждую подзадачу мы решаем только один раз, и к моменту, когда мы впервые с ней сталкиваемся, все необходимые для ее решения подзадачи уже решены.

Эти два подхода приводят к алгоритмам с одним и тем же асимптотическим временем работы, за исключением редких ситуаций, когда нисходящий подход в действительности не выполняет рекурсивное изучение всех возможных подзадач. Зачастую восходящий подход обладает лучшими константными множителями, поскольку при его применении оказываются меньшими накладные расходы, связанные с вызовами функций.

Вот как выглядит псевдокод нисходящей процедуры CUT-ROD с добавленным запоминанием.

```
MEMOIZED-CUT-ROD(p, n)
r[0..n] — новый массив
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

---

[3]В данном случае это не опечатка, по-английски этот термин пишется именно так. Как поясняют в примечании авторы книги, смысл не просто в том, чтобы запомнить информацию (memorize), а в том, чтобы вскоре ею воспользоваться, как памяткой (memo). — *Примеч. пер.*

Здесь основная процедура MEMOIZED-CUT-ROD инициализирует новый вспомогательный массив $r[0\,..\,n]$ значением $-\infty$, которое представляет собой удобный выбор для обозначения "неизвестно". (Известные значения прибыли всегда неотрицательны.) Затем она вызывает вспомогательную процедуру MEMOIZED-CUT-ROD-AUX.

Процедура MEMOIZED-CUT-ROD-AUX представляет собой версию с запоминанием предыдущей процедуры CUT-ROD. Сначала в строке 1 она проверяет, не известно ли уже искомое значение, и, если известно, возвращает его в строке 2. В противном случае в строках 3–7 процедура вычисляет искомое значение q обычным способом, в строке 8 сохраняет его в $r[n]$, а в строке 9 возвращает это значение вызвавшему коду.

Восходящая версия еще проще.

```
BOTTOM-UP-CUT-ROD(p, n)
1  r[0..n] — новый массив
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

При восходящем подходе динамического программирования процедура BOTTOM-UP-CUT-ROD использует естественное упорядочение подзадач: подзадача размером i "меньше" подзадачи размером j, если $i < j$. Таким образом, процедура решает подзадачи размером $j = 0, 1, \ldots, n$ в указанном порядке.

В строке 1 процедуры BOTTOM-UP-CUT-ROD создается новый массив $r[0\,..\,n]$, в котором хранятся решения подзадач, а в строке 2 элемент $r[0]$ инициализируется значением 0, поскольку стержень длиной 0 не приносит никакой прибыли. В строках 3–6 решается каждая подзадача размером j для $j = 1, 2, \ldots, n$ в порядке возрастания. Подход, используемый для решения задачи определенного размера j, тот же, что и применяемый в процедуре CUT-ROD, с тем отличием, что в строке 6 выполняется непосредственное обращение к элементу массива $r[j - i]$ вместо рекурсивного вызова для решения подзадачи размером $j - i$. В строке 7 выполняется сохранение в $r[j]$ решения подзадачи размером j. Наконец в строке 8 выполняется возврат $r[n]$, оптимального значения r_n.

Восходящая и нисходящая версии имеют одно и то же асимптотическое время работы. Из-за вложенной структуры циклов время работы процедуры BOTTOM-UP-CUT-ROD составляет $\Theta(n^2)$. Количества итераций внутреннего цикла **for** в строках 5 и 6 образуют арифметическую прогрессию. Время работы нисходящего двойника, процедуры MEMOIZED-CUT-ROD, также равно $\Theta(n^2)$, хотя это время может быть не столь очевидным. Поскольку рекурсивный вызов для решения ранее решенных задач немедленно возвращается, процедура MEMOIZED-CUT-ROD решает каждую подзадачу только один раз. Она решает подзадачи раз-

мером $0, 1, \ldots, n$. Для решения подзадачи размером n цикл **for** в строках 6 и 7 выполняет n итераций. Таким образом, общее количество итераций этого цикла **for** по всем рекурсивным вызовам MEMOIZED-CUT-ROD образует арифметическую прогрессию, что приводит к общему количеству итераций, равному $\Theta(n^2)$, как и в случае внутреннего цикла **for** процедуры BOTTOM-UP-CUT-ROD. (В действительности нами использована разновидность группового анализа, который будет подробно рассматриваться в разделе 17.1.)

Графы подзадач

При рассмотрении задачи динамического программирования следует найти множество решаемых подзадач и понять, как подзадачи зависят одна от другой.

Граф подзадач для задачи динамического программирования содержит интересующую нас информацию. На рис. 15.4 показан граф подзадач для задачи разрезания стержня при $n = 4$. Это ориентированный граф, содержащий по одной вершине для каждой из различных подзадач. Граф подзадач содержит дугу, идущую от вершины подзадачи x к вершине подзадачи y, если определение оптимального решения подзадачи x непосредственно включает поиск оптимального решения для подзадачи y. Например, граф подзадач содержит дугу, идущую от x к y, если нисходящая рекурсивная процедура для решения x непосредственно вызывает саму себя для решения y. Граф подзадач можно рассматривать как "уменьшенную" или "сжатую" версию дерева рекурсии нисходящего рекурсивного метода, в которой мы сливаем все узлы для одной и той же подзадачи в единую вершину и направляем все дуги от родительских узлов к дочерним.

Восходящий метод динамического программирования рассматривает вершины графа подзадач в том порядке, в котором решаются сами подзадачи, т.е. все подзадачи y, смежные данной подзадаче x, решаются до того, как мы приступим к решению подзадачи x. (Вспомним из раздела Б.4, что отношение смежности не обязано быть симметричным.) Используя терминологию главы 22, в восходящем алгоритме динамического программирования вершины графа подзадач

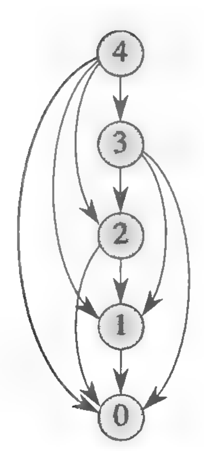


Рис. 15.4. Граф подзадач для задачи разрезания стержня при $n = 4$. Метки вершин указывают размеры соответствующих подзадач. Ориентированное ребро (x, y) указывает, что при решении подзадачи x необходимо решить подзадачу y. Этот граф представляет собой уменьшенную версию дерева на рис. 15.3, в которой все узлы на одном уровне слиты в единую вершину, а все ребра превращены в дуги, идущие от родительского узла к дочернему.

рассматриваются в порядке "обратной топологической сортировки", или "топологической сортировки транспозиции" (см. раздел 22.4) графа подзадач. Другими словами, никакая подзадача не рассматривается до тех пор, пока не будут решены все подзадачи, от которых она зависит. Аналогично, используя понятия из той же главы, мы можем рассматривать нисходящий метод (с запоминанием) динамического программирования как "поиск в глубину" в графе подзадач (см. раздел 22.3).

Размер графа подзадач $G = (V, E)$ может помочь в определении времени работы алгоритма динамического программирования. Поскольку каждая подзадача решается однократно, время работы алгоритма представляет собой сумму времен, необходимых для решения каждой подзадачи. Обычно время решения подзадачи пропорционально степени (количеству исходящих ребер) соответствующей вершины в графе подзадач, а количество подзадач равно числу вершин в графе подзадач. В этом распространенном случае время работы алгоритма динамического программирования линейно зависит от числа вершин и ребер.

Восстановление решения

Наше решение задачи разрезания стержня путем динамического программирования возвращает значение оптимального решения, но не само решение, которое должно иметь вид списка размеров частей. Можно расширить подход динамического программирования и записывать не только вычисленное оптимальное *значение* каждой подзадачи, но и *выбор*, который приводит к этому оптимальному значению. При наличии этой информации мы можем легко вывести оптимальное решение.

Вот как выглядит расширенная версия процедуры Bottom-Up-Cut-Rod, которая для каждого размера стержня j вычисляет не только максимальную прибыль r_j, но и оптимальный размер первой отрезаемой части s_j.

```
Extended-Bottom-Up-Cut-Rod(p, n)
 1  r[0..n] и s[0..n] — новые массивы
 2  r[0] = 0
 3  for j = 1 to n
 4      q = −∞
 5      for i = 1 to j
 6          if q < p[i] + r[j − i]
 7              q = p[i] + r[j − i]
 8              s[j] = i
 9      r[j] = q
10  return r и s
```

Эта процедура аналогична процедуре Bottom-Up-Cut-Rod, с тем отличием, что она создает массив s в строке 1 и обновляет $s[j]$ в строке 8, сохраняя оптимальный размер i первой отрезаемой части при решении подзадачи размером j.

Приведенная далее процедура получает таблицу цен p и размер стержня n и вызывает процедуру Extended-Bottom-Up-Cut-Rod для вычисления масси-

ва $s[1..n]$ оптимальных размеров первых частей, а затем выводит полный список размеров частей в оптимальном разрезании стержня длиной n.

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

В нашем примере вызов EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ вернет следующие массивы.

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Вызов PRINT-CUT-ROD-SOLUTION$(p, 10)$ выведет единственную часть длиной 10, но для $n = 7$ будут выведены размеры частей 1 и 6, соответствующие первому оптимальному разрезанию для r_7, приведенному ранее.

Упражнения

15.1.1

Покажите, что уравнение (15.4) следует из уравнения (15.3) и начального условия $T(0) = 1$.

15.1.2

Покажите с помощью контрпримера, что описанная далее жадная стратегия не всегда определяет оптимальный способ разрезания стержня. Определим ***плотность*** стержня длиной i как p_i/i, т.е. как стоимость единицы его длины. Жадная стратегия для стержня длиной n отрезает от стержня первую часть длиной i, где $1 \leq i \leq n$, имеющую максимальную плотность. Затем та же жадная стратегия применяется к оставшейся части длиной $n - i$.

15.1.3

Рассмотрим модификацию задачи разрезания стержня, в которой в дополнение к цене p_i каждого стержня добавляется фиксированная цена разреза c. Теперь прибыль, связанная с решением, представляет собой сумму цен частей стержня минус стоимость разрезов. Разработайте алгоритм динамического программирования для этой модифицированной задачи.

15.1.4

Модифицируйте процедуру MEMOIZED-CUT-ROD таким образом, чтобы она возвращала не только значение, но и фактическое решение задачи.

15.1.5

Числа Фибоначчи определяются рекуррентным соотношением (3.22). Разработайте алгоритм динамического программирования со временем работы $O(n)$ для вычисления n-го числа Фибоначчи. Изобразите граф подзадач. Сколько вершин и ребер имеет этот граф?

15.2. Перемножение цепочки матриц

Очередиой пример применения динамического программирования — алгоритм, позволяющий решить задачу о перемножении цепочки матриц. Пусть имеется последовательность (цепочка) $\langle A_1, A_2, \ldots, A_n \rangle$, состоящая из n матриц, и нужно вычислить их произведение

$$A_1 A_2 \cdots A_n . \tag{15.5}$$

Выражение (15.5) можно вычислить, используя в качестве подпрограммы стандартный алгоритм перемножения пар матриц. Однако сначала нужно расставить скобки, чтобы устранить все неоднозначности в порядке перемножения. Операция умножения матриц ассоциативна, так что любая расстановка скобок даст один и тот же результат. Порядок произведения матриц ***полностью определен скобками*** (fully parenthesized), если произведение является либо отдельной матрицей, либо взятым в скобки произведением двух подпоследовательностей матриц, в котором порядок перемножения полностью определен скобками. Например, если задана последовательность матриц $\langle A_1, A_2, A_3, A_4 \rangle$, то способ вычисления их произведения $A_1 A_2 A_3 A_4$ можно полностью определить с помощью скобок пятью разными способами:

$$\begin{aligned}
&(A_1(A_2(A_3A_4))) , \\
&(A_1((A_2A_3)A_4)) , \\
&((A_1A_2)(A_3A_4)) , \\
&((A_1(A_2A_3))A_4) , \\
&(((A_1A_2)A_3)A_4) .
\end{aligned}$$

От того, как расставлены скобки при перемножении последовательности матриц, может сильно зависеть время, затраченное на вычисление произведения. Сначала рассмотрим, как определить стоимость произведения двух матриц. Ниже приводится псевдокод стандартного алгоритма, который обобщает процедуру SQUARE-MATRIX-MULTIPLY из раздела 4.2. Атрибуты *rows* и *columns* означают количество строк и столбцов матрицы.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "несовместимые размеры матриц"
3  else C — новая матрица размером A.rows × B.columns
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

Матрицы A и B можно перемножать, только если они ***совместимы***: количество столбцов матрицы A должно совпадать с количеством строк матрицы B. Если A — это матрица размером $p \times q$, а B — матрица размером $q \times r$, то в результате их перемножения получится матрица C размером $p \times r$. Время вычисления матрицы C преимущественно определяется количеством произведений скаляров (далее в главе для краткости будем называть эту операцию скалярным умножением. — *Примеч. пер.*), которое выполняется в строке 8. Это количество равно pqr. Итак, стоимость умножения матриц будет выражаться в терминах количества умножений скалярных величин.

Чтобы проиллюстрировать, как расстановка скобок при перемножении нескольких матриц влияет на количество выполняемых операций, рассмотрим пример, в котором перемножаются три матрицы — $\langle A_1, A_2, A_3 \rangle$. Предположим, что размеры этих матриц равны 10×100, 100×5 и 5×50 соответственно. Перемножая матрицы в порядке, заданном выражением $((A_1A_2)A_3)$, необходимо выполнить $10 \cdot 100 \cdot 5 = 5000$ скалярных умножений, чтобы найти результат произведения A_1A_2 (при этом получится матрица размером 10×5), а затем — еще $10 \cdot 5 \cdot 50 = 2500$ скалярных умножений, чтобы умножить эту матрицу на матрицу A_3. Всего получается 7500 скалярных умножений. Если вычислять результат в порядке, заданном выражением $(A_1(A_2A_3))$, то сначала понадобится выполнить $100 \cdot 5 \cdot 50 = 25\,000$ скалярных умножений (при этом будет найдена матрица A_2A_3 размером 100×50), а затем еще $10 \cdot 100 \cdot 50 = 50\,000$ скалярных умножений, чтобы умножить A_1 на эту матрицу. Всего получается 75 000 скалярных умножений. Таким образом, для вычисления результата первым способом понадобится в 10 раз меньше времени.

Задачу о перемножении последовательности матриц (matrix-chain multiplication problem) можно сформулировать следующим образом: для заданной последовательности n матриц $\langle A_1, A_2, \ldots, A_n \rangle$, в которой матрица A_i, $i = 1, 2, \ldots, n$ имеет размер $p_{i-1} \times p_i$, с помощью скобок следует полностью определить порядок умножений в матричном произведении $A_1A_2 \cdots A_n$, при котором количество скалярных умножений сведется к минимуму.

Обратите внимание, что само перемножение матриц в задачу не входит. Наша цель — определить оптимальный порядок перемножения. Обычно время, затраченное на нахождение оптимального способа перемножения матриц, с лихвой окупается, когда выполняется само перемножение (как это было в рассмотрен-

ном примере, когда удалось обойтись всего 7500 скалярными умножениями вместо 75 000).

Подсчет количества способов расстановки скобок

Прежде чем приступить к решению задачи об умножении последовательности матриц методами динамического программирования, заметим, что исчерпывающая проверка всех возможных вариантов расстановки скобок не является эффективным алгоритмом ее решения. Обозначим через $P(n)$ количество различных способов расстановки скобок в последовательности, состоящей из n матриц. Если $n = 1$, то матрица всего одна, поэтому скобки в матричном произведении можно расставить всего одним способом. Если $n \geq 2$, то произведение последовательности матриц, в котором порядок перемножения полностью определен скобками, является произведением двух таких произведений подпоследовательностей матриц, в которых порядок перемножения также полностью определен скобками. Разбиение на подпоследовательности может производиться на границе k- и $(k+1)$-й матриц для любого $k = 1, 2, \ldots, n-1$. В результате получаем рекуррентное соотношение

$$P(n) = \begin{cases} 1\ , & \text{если } n = 1\ , \\ \sum\limits_{k=1}^{n-1} P(k)P(n-k)1\ , & \text{если } n \geq 2\ . \end{cases} \qquad (15.6)$$

В задаче 12.4 предлагается показать, что решением аналогичного рекуррентного соотношения является последовательность ***чисел Каталана*** (Catalan numbers), возрастающая как $\Omega(4^n/n^{3/2})$. Более простое упражнение (упр. 15.2.3) заключается в том, чтобы показать, что решение рекуррентного соотношения (15.6) ведет себя, как $\Omega(2^n)$. Таким образом, количество вариантов расстановки скобок экспоненциально увеличивается с ростом n, и метод прямого перебора всех вариантов не подходит для определения оптимальной стратегии расстановки скобок в матричном произведении.

Применение динамического программирования

Для вычисления оптимальной расстановки скобок при перемножении последовательности матриц мы применим динамическое программирование. Для этого мы должны следовать последовательности из четырех этапов, описанной в начале данной главы.

1. Описание структуры оптимального решения.
2. Определение значения, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.

4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

Мы последовательно пройдем все эти этапы, ясно демонстрируя их применение к данной задаче.

Этап 1. Структура оптимальной расстановки скобок

Первый этап применения парадигмы динамического программирования — найти оптимальную вспомогательную подструктуру, а затем с ее помощью сконструировать оптимальное решение задачи по оптимальным решениям подзадач. В рассматриваемой задаче этот этап можно осуществить следующим образом. Обозначим для удобства результат перемножения матриц $A_i A_{i+1} \cdots A_j$ через $A_{i..j}$, где $i \leq j$. Заметим, что если задача нетривиальна, т.е. $i < j$, то любой способ расстановки скобок в произведении $A_i A_{i+1} \cdots A_j$ разбивает это произведение между матрицами A_k и A_{k+1}, где k — целое, удовлетворяющее условию $i \leq k < j$. Таким образом, при некотором k сначала выполняется вычисление матриц $A_{i..k}$ и $A_{k+1..j}$, а затем они умножаются друг на друга, в результате чего получается произведение $A_{i..j}$. Стоимость, соответствующая этому способу расстановки скобок, равна сумме стоимости вычисления матрицы $A_{i..k}$, стоимости вычисления матрицы $A_{k+1..j}$ и стоимости вычисления их произведения.

Ниже описывается оптимальная вспомогательная подструктура для данной задачи. Предположим, что в результате оптимальной расстановки скобок последовательность $A_i A_{i+1} \cdots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}. Тогда расстановка скобок в "префиксной" подпоследовательности $A_i A_{i+1} \cdots A_k$ также должна быть оптимальной. Почему? Если бы существовал более экономный способ расстановки скобок в последовательности $A_i A_{i+1} \cdots A_k$, то его применение позволило бы перемножить матрицы $A_i A_{i+1} \cdots A_j$ еще эффективнее, что противоречит предположению об оптимальности первоначальной расстановки скобок. Аналогично можно прийти к выводу, что расстановка скобок в подпоследовательности матриц $A_{k+1} A_{k+2} \cdots A_j$, возникающей в результате оптимальной расстановки скобок в последовательности $A_i A_{i+1} \cdots A_j$, также должна быть оптимальной.

Теперь с помощью нашей оптимальной вспомогательной подструктуры покажем, что оптимальное решение задачи можно составить из оптимальных решений подзадач. Мы уже убедились, что для решения любой нетривиальной задачи об оптимальном произведении последовательности матриц всю последовательность необходимо разбить на подпоследовательности и что каждое оптимальное решение содержит в себе оптимальные решения подзадач. Другими словами, решение полной задачи об оптимальном перемножении последовательности матриц можно построить путем разбиения этой задачи на две подзадачи — оптимальную расстановку скобок в подпоследовательностях $A_i A_{i+1} \cdots A_k$ и $A_{k+1} A_{k+2} \cdots A_j$. После этого находятся оптимальные решения подзадач, из которых затем составляется оптимальное решение полной задачи. Необходимо убедиться, что при поиске способа перемножения матриц учитываются все возможные варианты разбиения —

только в этом случае можно быть уверенным, что найденное решение будет глобально оптимальным.

Этап 2. Рекурсивное решение

Далее рекурсивно определим стоимость оптимального решения в терминах оптимальных решений подзадач. В задаче о перемножении последовательности матриц в качестве подзадачи выбирается задача об оптимальной расстановке скобок в подпоследовательности $A_iA_{i+1}\cdots A_j$ при $1 \leq i \leq j \leq n$. Путь $m[i,j]$ — минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Тогда в полной задаче минимальная стоимость матрицы $A_{1..n}$ равна $m[1,n]$.

Рекурсивно определить величину $m[i,j]$ можно следующим образом. Если $i = j$, то задача становится тривиальной: последовательность состоит всего из одной матрицы $A_{i..i} = A_i$, и для вычисления произведения матриц не нужно выполнять никаких скалярных умножений. Таким образом, при $i = 1, 2, \ldots, n$ имеем $m[i,i] = 0$. Чтобы вычислить $m[i,j]$ при $i < j$, воспользуемся свойством подструктуры оптимального решения, исследованным на этапе 1. Предположим, что в результате оптимальной расстановки скобок последовательность $A_iA_{i+1}\cdots A_j$ разбивается между матрицами A_k и A_{k+1}, где $i \leq k < j$. Тогда величина $m[i,j]$ равна минимальной стоимости вычисления частных произведений $A_{i..k}$ и $A_{k+1..j}$ плюс стоимость умножения этих матриц друг на друга. Если вспомнить, что каждая матрица A_i имеет размеры $p_{i-1} \times p_i$, то нетрудно понять, что для вычисления произведения матриц $A_{i..k}A_{k+1..j}$ понадобится $p_{i-1}p_kp_j$ скалярных умножений. Таким образом, получаем

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \ .$$

В этом рекурсивном уравнении предполагается, что значение k известно, но на самом деле это не так. Для выбора этого значения всего имеется $j - i$ возможностей, а именно — $k = i, i+1, \ldots, j-1$. Поскольку в оптимальной расстановке скобок необходимо использовать одно из этих значений k, все, что нужно сделать, — проверить все возможности и выбрать среди них лучшую. Таким образом, рекурсивное определение оптимальной расстановки скобок в произведении $A_iA_{i+1}\cdots A_j$ принимает вид

$$m[i,j] = \begin{cases} 0 \ , & \text{если } i = j \ , \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} \ , & \text{если } i < j \ . \end{cases} \qquad (15.7)$$

Величины $m[i,j]$ равны стоимостям оптимальных решений подзадач. Чтобы легче было проследить за процессом построения оптимального решения, обозначим через $s[i,j]$ значение k, в котором последовательность $A_iA_{i+1}\cdots A_j$ разбивается на две подпоследовательности в процессе оптимальной расстановки скобок. Таким образом, величина $s[i,j]$ равна значению k, такому, что $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Этап 3. Вычисление оптимальных стоимостей

На данном этапе не составляет труда написать на основе рекуррентного соотношения (15.7) рекурсивный алгоритм для вычисления минимальной стоимости $m[1,n]$ для произведения $A_1A_2 \cdots A_n$. Однако в разделе 15.3 мы сможем убедиться, что время работы этого алгоритма экспоненциально зависит от n, что ничем не лучше метода прямого перебора, при котором проверяется каждый способ расстановки скобок в произведении.

Важное наблюдение, которое можно сделать на данном этапе, заключается в том, что у нас относительно мало различных подзадач: по одной для каждого выбора величин i и j, удовлетворяющих неравенству $1 \leq i \leq j \leq n$, т.е. всего $\binom{n}{2} + n = \Theta(n^2)$. В рекурсивном алгоритме каждая подзадача может неоднократно встречаться в разных ветвях рекурсивного дерева. Такое свойство перекрытия подзадач — вторая отличительная черта применимости метода динамического программирования (первая отличительная черта — наличие оптимальной подструктуры).

Вместо того чтобы рекурсивно решать рекуррентное соотношение (15.12), выполним этап 3 парадигмы динамического программирования и вычислим оптимальную стоимость путем построения таблицы в восходящем направлении. В описанной ниже процедуре предполагается, что размеры матриц A_i равны $p_{i-1} \times p_i$ $(i = 1, 2, \ldots, n)$. Входные данные представляют собой последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$; длина данной последовательности равна $length[p] = n+1$. В процедуре используется вспомогательная таблица $m[1..n, 1..n]$ для хранения стоимостей $m[i,j]$ и вспомогательная таблица $s[1..n, 1..n]$, в которую заносятся индексы k, при которых достигаются оптимальные стоимости $m[i,j]$. Таблица s будет использоваться при построении оптимального решения.

Вместо рекурсивного вычисления рекуррентного соотношения (15.7) мы вычисляем оптимальную стоимость с помощью табличного восходящего подхода. (Соответствующий нисходящий подход с запоминанием будет представлен в разделе 15.3.)

Мы реализуем табличный восходящий подход в процедуре MATRIX-CHAIN-ORDER, приведенной ниже. В этой процедуре предполагается, что матрица A_i имеет размер $p_{i-1} \times p_i$ для $i = 1, 2, \ldots, n$. Ее входными данными является последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$, где $p.length = n + 1$. Процедура использует вспомогательную таблицу $m[1\,..\,n, 1\,..\,n]$ для хранения стоимостей $m[i,j]$ и другую вспомогательную таблицу, $s[1\,..\,n-1, 2\,..\,n]$, в которую записывается, для какого индекса k достигается оптимальная стоимость при вычислении $m[i,j]$. Таблица s будет использоваться при построении оптимального решения.

Чтобы корректно реализовать восходящий подход, необходимо определить, с помощью каких записей таблицы будут вычисляться величины $m[i,j]$. Из уравнения (15.7) видно, что стоимость $m[i,j]$ вычисления произведения последовательности $j - i + 1$ матриц зависит только от стоимости вычисления последовательностей матриц, содержащих менее $j - i + 1$ матриц. Другими словами, при $k = i, i+1, \ldots, j-1$ матрица $A_{i..k}$ представляет собой произведение $k - i + 1 < j - i + 1$ матриц, а матрица $A_{k+1..j}$ — произведение $j - k < j - i + 1$ матриц. Таким образом, в ходе выполнения алгоритма следует организовать за-

полнение таблицы m в порядке, соответствующем решению задачи о расстановке скобок в последовательностях матриц возрастающей длины. В случае подзадачи оптимальной расстановки скобок в цепочке $A_i A_{i+1} \cdots A_j$ мы рассматриваем размер подзадачи как равный длине $j - i + 1$ цепочки.

```
MATRIX-CHAIN-ORDER(p)
1  n = p.length − 1
2  m[1..n, 1..n] и s[1..n − 1, 2..n] — новые таблицы
3  for i = 1 to n
4      m[i, i] = 0
5  for l = 2 to n                 // l — длина цепочки
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          m[i, j] = ∞
9          for k = i to j − 1
10             q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11             if q < m[i, j]
12                 m[i, j] = q
13                 s[i, j] = k
14 return m и s
```

Сначала в этом алгоритме (строки 3 и 4) выполняется инициализация $m[i, i] = 0$ для $i = 1, 2, \ldots, n$ (минимальные стоимости для последовательностей единичной длины). Затем в первой итерации цикла **for** в строках 5–13 с помощью рекуррентного соотношения (15.7) вычисляются величины $m[i, i+1]$ при $i = 1, 2, \ldots, n-1$ (минимальные стоимости для последовательностей длиной $l = 2$). При втором проходе этого цикла вычисляются величины $m[i, i+2]$ при $i = 1, 2, \ldots, n-2$ (минимальные стоимости для последовательностей длиной $l = 3$) и т.д. На каждом этапе вычисляемые в строках 10–13 величины $m[i, j]$ зависят только от уже вычисленных и занесенных в таблицу значений $m[i, k]$ и $m[k+1, j]$.

На рис. 15.5 описанный выше процесс проиллюстрирован для цепочки, состоящей из $n = 6$ матриц. Поскольку величины $m[i, j]$ определены только для $i \le j$, используется только часть таблицы m, расположенная над ее главной диагональю. Таблица на рисунке повернута так, чтобы ее главная диагональ была расположена горизонтально. В нижней части рисунка приведен список матриц, входящих в последовательность. На этой схеме легко найти минимальную стоимость $m[i, j]$ перемножения подцепочки матриц $A_i A_{i+1} \cdots A_j$. Она находится на пересечении линий, идущих от матрицы A_i вправо и вверх и от матрицы A_j — влево и вверх. В каждой горизонтальной строке таблицы содержатся стоимости перемножения подцепочек, состоящих из одинакового количества матриц. В процедуре MATRIX-CHAIN-ORDER строки вычисляются снизу вверх, а элементы в каждой строке — слева направо. Величина $m[i, j]$ вычисляется с помощью произведений $p_{i-1} p_k p_j$ для $k = i, i+1, \ldots, j-1$ и всех величин внизу слева и внизу справа от $m[i, j]$.

Несложный анализ структуры вложенных циклов в процедуре MATRIX-CHAIN-ORDER показывает, что время ее работы составляет $O(n^3)$. Глубина вло-

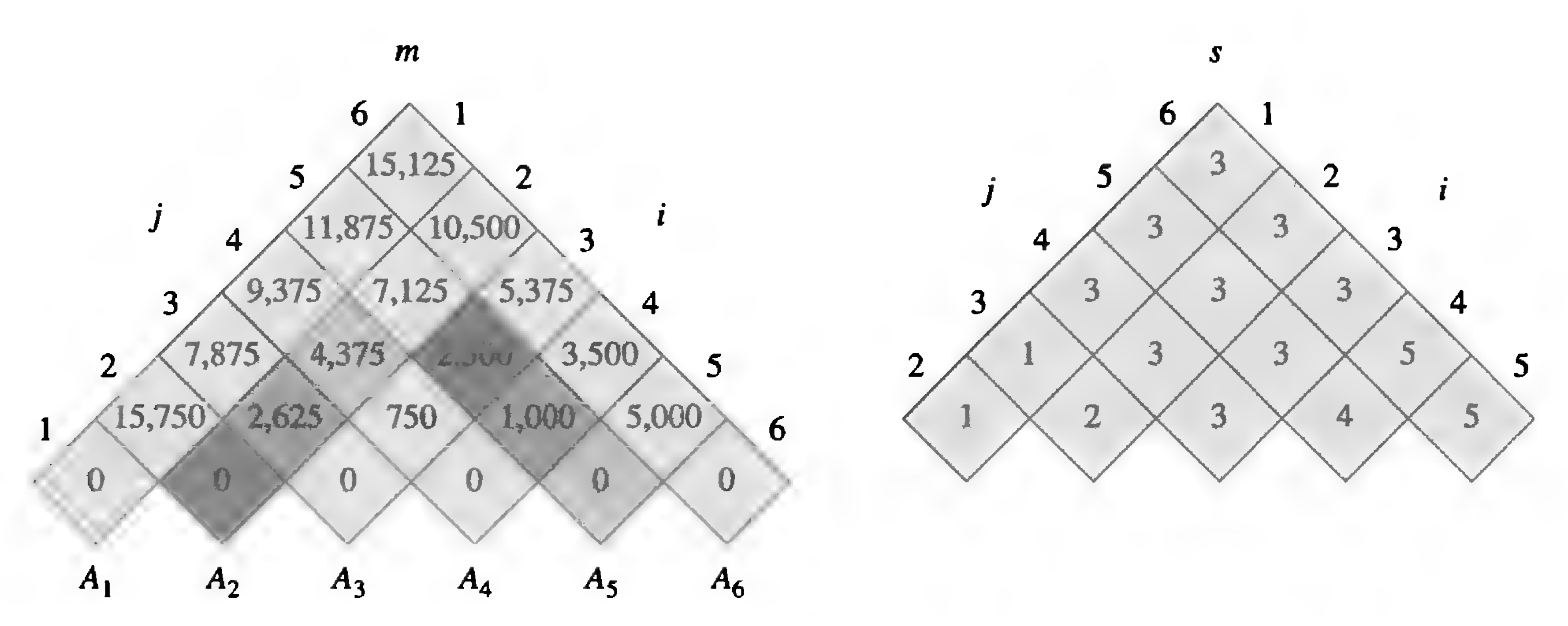


Рис. 15.5. Таблицы m и s, вычисляемые процедурой MATRIX-CHAIN-ORDER для $n = 6$ и следующих размерностей матриц.

Матрица	A_1	A_2	A_3	A_4	A_5	A_6
Размерность	30×35	35×15	15×5	5×10	10×20	20×25

Таблицы повернуты таким образом, чтобы главная диагональ располагалась горизонтально. В таблице m используются только главная диагональ и верхний треугольник, а в таблице s — только верхний треугольник, без главной диагонали. Минимальное количество скалярных умножений для вычисления произведений шести матриц равно $m[1,6] = 15\,125$. Более темным цветом выделены пары, совместно используемые в строке 10 при вычислении

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13\,000\,, \\ m[2,3] + m[4,5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11\,375 \end{cases}$$
$$= 7125\,.$$

жения циклов равна трем, а индексы в каждом из них (l, i и k) принимают не более $n-1$ значений. В упр. 15.2.5 предлагается показать, что время работы этого алгоритма фактически равно $\Omega(n^3)$. Для хранения таблиц m и s требуется объем, равный $\Theta(n^2)$. Таким образом, процедура MATRIX-CHAIN-ORDER намного эффективнее, чем метод перебора и проверки всевозможных способов расстановки скобок, время работы которого экспоненциально зависит от количества перемножаемых матриц.

Этап 4. Построение оптимального решения

Несмотря на то что в процедуре MATRIX-CHAIN-ORDER определяется оптимальное количество скалярных произведений, необходимых для вычисления произведения последовательности матриц, в нем не показано, как именно перемножаются матрицы. Оптимальное решение несложно построить с помощью информации, хранящейся в таблице $s[1\,..\,n-1, 2\,..\,n]$. В каждом элементе $s[i,j]$ хранится значение индекса k, где при оптимальной расстановке скобок в последовательности $A_iA_{i+1}\cdots A_j$ выполняется разбиение. Таким образом, нам известно, что оптимальное вычисление произведения матриц $A_{1..n}$ выглядит как $A_{1..s[1,n]}A_{s[1,n]+1..n}$. Все предшествующие произведения матриц можно вычислить рекурсивно, поскольку элемент $s[1, s[1,n]]$ определяет матричное умножение, вы-

полняемое последним при вычислении $A_{1..s[1,n]}$, а $s[s[1,n]+1,n]$ — последнее умножение при вычислении $A_{s[1,n]+1..n}$. Приведенная ниже рекурсивная процедура выводит оптимальный способ расстановки скобок в последовательности матриц $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ по таблице s, полученной в результате работы процедуры MATRIX-CHAIN-ORDER, и по индексам i и j. Первоначальный вызов процедуры PRINT-OPTIMAL-PARENS$(s, 1, n)$ выводит оптимальную расстановку скобок в последовательности $\langle A_1, A_2, \ldots, A_n \rangle$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

В примере, проиллюстрированном на рис. 15.5, вызов процедуры PRINT-OPTIMAL-PARENS$(s, 1, 6)$ дает строку $((A_1(A_2A_3))((A_4A_5)A_6))$.

Упражнения

15.2.1

Найдите оптимальную расстановку скобок в произведении последовательности матриц, размерности которых равны $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2.2

Разработайте рекурсивный алгоритм MATRIX-CHAIN-MULTIPLY(A, s, i, j), в котором оптимальным образом вычисляется произведение заданной последовательности матриц $\langle A_1, A_2, \ldots, A_n \rangle$. На вход этого алгоритма, кроме того, поступают индексы i и j, а также таблица s, вычисленная с помощью процедуры MATRIX-CHAIN-ORDER. (Начальный вызов этой процедуры выглядит следующим образом: MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.)

15.2.3

Покажите с помощью метода подстановок, что решением рекуррентного соотношения (15.6) является $\Omega(2^n)$.

15.2.4

Опишите граф подзадач для перемножения цепочки матриц для входной цепочки длиной n. Сколько вершин в этом графе? Сколько в нем ребер, и что они собой представляют?

15.2.5

Пусть $R(i,j)$ — количество обращений к элементу матрицы $m[i,j]$, которые выполняются в ходе вычисления других элементов этой матрицы в процедуре MATRIX-CHAIN-ORDER. Покажите, что полное количество обращений ко всем

элементам равно

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3} .$$

(*Указание:* для решения может оказаться полезным уравнение (А.3).)

15.2.6

Покажите, что в полной расстановке скобок в n-элементном выражении используется ровно $n - 1$ пар скобок.

15.3. Элементы динамического программирования

Несмотря на рассмотренные в предыдущем разделе два примера, в которых применялся метод динамического программирования, возможно, вам все еще не совсем понятно, когда применим этот метод. В каких случаях задачу можно решать с помощью динамического программирования? В данном разделе рассматриваются два ключевых ингредиента, которые должны быть присущи задаче оптимизации, чтобы к ней можно было применить метод динамического программирования: наличие оптимальной подструктуры и перекрывающихся подзадач. Мы также более полно рассмотрим метод запоминания (memoization) и изучим, каким образом он позволяет воспользоваться преимуществами перекрывающихся подзадач при нисходящем рекурсивном подходе.

Оптимальная подструктура

Первый шаг решения задачи оптимизации методом динамического программирования состоит в том, чтобы охарактеризовать структуру оптимального решения. Напомним, что ***оптимальная подструктура*** проявляется в задаче в том случае, если в ее оптимальном решении содержатся оптимальные решения подзадач. Если в задаче выявлена оптимальная подструктура, это служит веским аргументом в пользу того, что к ней может быть применим метод динамического программирования. (Однако наличие этого свойства может также свидетельствовать о применимости жадных алгоритмов; см. главу 16.) В динамическом программировании оптимальное решение задачи строится из оптимальных решений подзадач. Следовательно, необходимо убедиться в том, что в число рассматриваемых подзадач входят те, которые используются в оптимальном решении.

Оптимальная подструктура была обнаружена в обеих задачах, которые до этого были исследованы в настоящей главе. В разделе 15.1 было установлено, что оптимальное разрезание стержня длиной n (если разрезание вообще имеет место) включает оптимальные разрезания частей, появившихся в результате первого разреза. В разделе 15.2 было продемонстрировано, что в оптимальную расстановку скобок, в результате которой последовательность матриц $A_i A_{i+1} \cdots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}, входят оптимально-

ные решения задач о расстановке скобок в подпоследовательностях $A_i A_{i+1} \cdots A_k$ и $A_{k+1} A_{k+2} \cdots A_j$.

Можно видеть, что поиск оптимальной подструктуры происходит по общему образцу.

1. На этапе 1 следует показать, что в процессе решения задачи приходится делать выбор. В рассмотренных примерах это был выбор начального разреза стержня или выбор индекса, при котором разбивается последовательность матриц. После выбора остается решить одну или несколько подзадач.

2. На этом этапе мы исходим из того, что для поставленной задачи делается выбор, ведущий к оптимальному решению. Пока что не рассматривается, как именно следует делать выбор, а просто предполагается, что он найден.

3. Исходя из заданного выбора определяется, какие подзадачи получаются и как лучше охарактеризовать получающееся в результате пространство подзадач.

4. Показывается, что решения подзадач, возникающих в ходе оптимального решения задачи, сами должны быть оптимальными. Это делается методом "от противного": предполагая, что решение каждой подзадачи не оптимально, приходим к противоречию. В частности, путем "вырезания" неоптимального решения подзадачи и "вставки" оптимального демонстрируется, что можно получить лучшее решение исходной задачи, а это противоречит предположению о том, что уже имеется оптимальное решение. Если подзадач несколько, они обычно настолько похожи, что описанный выше способ рассуждения, примененный к одной из подзадач, легко модифицируется для остальных.

Характеризуя пространство подзадач, постарайтесь придерживаться такого практического правила: попытайтесь сделать так, чтобы это пространство было как можно проще, а затем расширьте его до необходимых пределов. Например, пространство в подзадачах, возникающих в задаче о разрезании стержня, было образовано задачами оптимального разрезания стержня длиной i для каждого размера i. Это подпространство оказалось вполне подходящим, и не возникло необходимости его расширять.

Теперь предположим, что в задаче о перемножении цепочки матриц предпринимается попытка ограничить пространство подзадач произведениями вида $A_1 A_2 \cdots A_j$. Как и ранее, оптимальная расстановка скобок должна разбивать произведение между матрицами A_k и A_{k+1} для некоторого индекса $1 \leq k < j$. Если k не всегда равно $j - 1$, мы обнаружим, что возникают подзадачи вида $A_1 A_2 \cdots A_k$ и $A_{k+1} A_{k+2} \cdots A_j$ и что последняя подзадача не является подзадачей вида $A_1 A_2 \cdots A_j$. В этой задаче необходима возможность изменения обоих концов последовательности, т.е. чтобы в подзадаче $A_i A_{i+1} \cdots A_j$ могли меняться оба индекса — и i, и j.

Оптимальная подструктура изменяется в области определения задачи в двух аспектах.

1. Количество подзадач, которые используются при оптимальном решении исходной задачи.

2. Количество выборов, возникающих при определении того, какая подзадача (подзадачи) используется в оптимальном решении.

В оптимальном решении задачи о разрезании стержня длиной n используется только одна подзадача размером $n - i$, но для поиска оптимального решения необходимо рассмотрение n вариантов значения i. Перемножение подцепочки матриц $A_i A_{i+1} \cdots A_j$ служит примером с двумя подзадачами и $j - i$ вариантами выбора. Если задана матрица A_k, у которой происходит разбиение произведения, то возникают две подзадачи — расстановка скобок в подпоследовательностях $A_i A_{i+1} \cdots A_k$ и $A_{k+1} A_{k+2} \cdots A_j$, причем для *каждой* из них необходимо найти оптимальное решение. Как только оптимальные решения подзадач найдены, выбирается один из $j - i$ вариантов индекса k.

Если говорить неформально, время работы алгоритма динамического программирования зависит от произведения двух множителей: общего количества подзадач и количества вариантов выбора, возникающих в каждой подзадаче. При разрезании стержня всего у нас возникало $\Theta(n)$ подзадач и не более n вариантов выбора, что дает время работы $O(n^2)$. При перемножении матриц всего возникало $\Theta(n^2)$ подзадач, и в каждой из них — не более $n - 1$ вариантов выбора, поэтому время работы в этом случае равно $O(n^3)$ (на самом деле — $\Theta(n^3)$ согласно упр. 15.2.5).

Обычно граф подзадач представляет собой альтернативный путь выполнения этого анализа. Каждая вершина соответствует подзадаче, а варианты выбора для каждой подзадачи представлены ребрами, входящими в соответствующую вершину. Вспомним, что в задаче о разрезании стержня граф имел n вершин и не более n ребер на одну вершину, что дает время работы $O(n^2)$. Если изобразить граф подзадач для перемножения цепочки матриц, то он будет иметь $\Theta(n^2)$ вершин, а каждая вершина — степень не выше $n-1$, что всего дает $O(n^3)$ вершин и ребер.

Оптимальная подструктура часто используется в динамическом программировании в восходящем направлении. Другими словами, сначала находятся оптимальные решения подзадач, после чего определяется оптимальное решение поставленной задачи. Поиск оптимального решения задачи влечет за собой необходимость выбора одной из подзадач, которые будут использоваться в решении полной задачи. Стоимость решения задачи обычно определяется как сумма стоимостей решений подзадач и стоимости, которая затрачивается на определение правильного выбора. Например, в задаче о разрезании стержня мы сначала решали подзадачи оптимальных способов разрезания стержней длиной i для $i = 0, 1, \ldots, n-1$, а затем определяли, какая подзадача дает оптимальное решение для стержня длиной n, используя уравнение (15.2). Стоимость, которая относится к самому выбору, представляет собой член p_i в уравнении (15.2). В задаче о перемножении цепочки матриц сначала был определен оптимальный способ расстановки скобок в подцепочках $A_i A_{i+1} \cdots A_j$, а затем была выбрана матрица A_k, у которой выполняется разбиение произведения. Стоимость, которая относится к самому выбору, выражается членом $p_{i-1} p_k p_j$.

В главе 16 рассматриваются жадные алгоритмы, имеющие много общего с динамическим программированием. В частности, задачи, к которым применимы жадные алгоритмы, обладают оптимальной подструктурой. Одно из характер-

ных различий между жадными алгоритмами и динамическим программированием заключается в том, что в жадных алгоритмах оптимальная подструктура используется в нисходящем направлении. Вместо того чтобы находить оптимальные решения подзадач с последующим выбором одного из возможных вариантов, в жадных алгоритмах сначала делается выбор, который выглядит наилучшим на текущий момент, а затем решается возникшая в результате подзадача. При этом мы не беспокоимся о решении всех возможных меньших связанных подзадач. Удивительно, но в некоторых случаях данная стратегия работает!

Некоторые тонкости

Следует быть особенно внимательным в отношении вопроса о применимости оптимальной подструктуры, когда она отсутствует в задаче. Рассмотрим две задачи, в которых имеются ориентированный граф $G = (V, E)$ и вершины $u, v \in V$.

Задача о кратчайшем невзвешенном пути.[4] Нужно найти простой путь от вершины u к вершине v, состоящий из минимального количества ребер. Этот путь обязан быть простым, поскольку в результате удаления из него цикла получается путь, состоящий из меньшего количества ребер.

Задача о длиннейшем невзвешенном пути. Нужно найти простой путь от вершины u к вершине v, состоящий из максимального количества ребер. Требование простоты весьма важно, поскольку в противном случае можно проходить по одному и тому же циклу сколько угодно раз, получая в результате путь, состоящий из произвольно большого количества ребер.

В задаче о кратчайшем пути возникает оптимальная подструктура. Это можно показать с помощью таких рассуждений. Предположим, что $u \neq v$, так что задача является нетривиальной. В этом случае любой путь p от u к v должен содержать промежуточную вершину, скажем, w. (Заметим, что вершина w может совпадать с вершиной u или v.) Поэтому путь $u \overset{p}{\leadsto} v$ можно разложить на подпути $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$. Очевидно, что количество ребер, входящих в путь p, равно сумме числа ребер в путях p_1 и p_2. Мы утверждаем, что если путь p от вершины u к вершине v оптимальный (т.е. кратчайший), то p_1 должен быть кратчайшим путем от вершины u к вершине w. Почему? Аргумент такой: если бы существовал другой путь, соединяющий вершины u и v и состоящий из меньшего количества ребер, чем p_1 (скажем, p'_1), то можно было бы вырезать путь p_1 и вставить путь p'_1, в результате чего получился бы путь $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$, состоящий из меньшего количества ребер, чем путь p. А это противоречит предположению об оптимальности пути p. Аналогично p_2 должен быть кратчайшим путем от вершины w к вершине v. Таким образом, кратчайший путь от вершины u к вершине v можно найти, рассмотрев все промежуточные вершины w, найдя кратчайший путь от вершины u к вер-

---

[4]Термин "невзвешенный" используется, чтобы отличать эту задачу от той, в ходе решения которой находится кратчайший путь на взвешенных ребрах и с которой мы ознакомимся в главах 24 и 25. Для решения задачи о невзвешенном пути можно использовать метод поиска в ширину, описанный в главе 22.

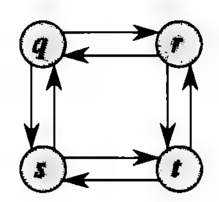


Рис. 15.6. Ориентированный граф, демонстрирующий, что задача поиска длиннейшего простого пути в невзвешенном ориентированном графе не имеет оптимальной подструктуры. Путь $q \to r \to t$ является наидлиннейшим простым путем от q к t, но ни подпуть $q \to r$ не является длиннейшим простым путем от q к r, ни подпуть $r \to t$ не является длиннейшим простым путем от r к t.

шине w и кратчайший путь от вершины w к вершине v и выбрав промежуточную вершину w, через которую весь путь окажется кратчайшим. В разделе 25.2 один из вариантов этой оптимальной подструктуры используется для поиска кратчайшего пути между всеми парами вершин во взвешенном ориентированном графе.

Напрашивается предположение о том, что в задаче поиска самого длинного простого невзвешенного пути также проявляется оптимальная подструктура. В конце концов, если разложить самый длинный простой путь $u \overset{p}{\leadsto} v$ на подпути $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, то разве не должен путь p_1 быть самым длинным простым путем от вершины u к вершине w, а путь p_2 — самым длинным простым путем от вершины w к вершине v? Оказывается, нет! Пример такой ситуации приведен на рис. 15.6. Рассмотрим путь $q \to r \to t$, который является самым длинным простым путем от вершины q к вершине t. Является ли путь $q \to r$ самым длинным путем от вершины q к вершине r? Нет, поскольку простой путь $q \to s \to t \to r$ длиннее. Является ли путь $r \to t$ самым длинным путем от вершины r к вершине t? Снова нет, поскольку простой путь $r \to q \to s \to t$ длиннее.

Этот пример демонстрирует, что в задаче о самых длинных простых путях не только отсутствует оптимальная подструктура, но и не всегда удается составить "законное" решение задачи из решений подзадач. Если сложить самые длинные простые пути $q \to s \to t \to r$ и $r \to q \to s \to t$, то получится путь $q \to s \to t \to r \to q \to s \to t$, который не является простым. Итак, на деле оказывается, что в задаче о поиске самого длинного невзвешенного пути не возникает никаких оптимальных подструктур. Для этой задачи до сих пор не найдено ни одного эффективного алгоритма, работающего по принципу динамического программирования. Фактически это NP-полная задача, что означает — как будет показано в главе 34, — что вряд ли ее можно решить в течение полиномиального времени.

Почему подструктура самого длинного простого пути так отличается от подструктуры самого короткого пути? Несмотря на то что в решениях задач о поиске и самого короткого, и самого длинного пути возникают две подзадачи, эти подзадачи при определении самого длинного пути не являются ***независимыми***, в то время как в задаче о кратчайшем пути они независимы. Что подразумевается под независимостью подзадач? Подразумевается, что решение одной подзадачи не влияет на решение другой подзадачи, возникающей в той же задаче. В примере, проиллюстрированном на рис. 15.6, рассматривается задача о поиске самого длинного простого пути между вершинами q и t, в которой возникают две подзадачи: определение самых длинных простых путей между вершинами q и r и между

вершинами r и t. Для первой из этих подзадач выбирается путь $q \to s \to t \to r$, в котором используются вершины s и t. Во второй подзадаче мы не сможем больше использовать эти вершины, поскольку в процессе комбинирования решений этих двух подзадач получился бы путь, который не является простым. Если во второй задаче нельзя использовать вершину t, то ее вообще невозможно решить, поскольку эта вершина должна быть в найденном пути, и это не та вершина, в которой "соединяются" решения подзадач (такой вершиной является r). Использование вершин s и t в решении одной подзадачи приводит к невозможности их применения в решении другой подзадачи. Однако для ее решения необходимо использовать хотя бы одну из них, а в оптимальное решение данной подзадачи входят обе эти вершины. Поэтому эти подзадачи не являются независимыми. Другими словами, использование ресурсов в решении одной подзадачи (в качестве ресурсов выступают вершины) делает их недоступными в другой подзадаче.

Почему же подзадачи остаются независимыми при поиске самого короткого пути? Ответ такой: по самой природе поставленной задачи возникающие в ней подзадачи не используют одни и те же ресурсы. Утверждается, что если вершина w находится на кратчайшем пути p от вершины u к вершине v, то можно соединить *любой* кратчайший путь $u \overset{p_1}{\leadsto} w$ с *любым* кратчайшим путем $w \overset{p_2}{\leadsto} v$, получив в результате самый короткий путь от вершины u к вершине v. Мы уверены в том, что пути p_1 и p_2 не содержат ни одной общей вершины, кроме w. Почему? Предположим, что имеется некоторая вершина $x \neq w$, принадлежащая путям p_1 и p_2, так что путь p_1 можно разложить как $u \overset{p_{ux}}{\leadsto} x \leadsto w$, а путь p_2 — как $w \leadsto x \overset{p_{xv}}{\leadsto} v$. В силу оптимальной подструктуры этой задачи количество ребер в пути p равно сумме количеств ребер в путях p_1 и p_2. Предположим, что путь p содержит e ребер. Теперь построим путь $p' = u \overset{p_{ux}}{\leadsto} x \overset{p_{xv}}{\leadsto} v$ от вершины u к вершине v. В этом пути содержится не более $e-2$ вершин, что противоречит предположению о том, что путь p — кратчайший. Таким образом, мы убедились, что подзадачи, возникающие в задаче поиска кратчайшего пути, являются независимыми.

Подзадачи, возникающие в задачах, которые рассматриваются в разделах 15.1 и 15.2, являются независимыми. При перемножении цепочки матриц подзадачи заключались в перемножении подцепочек $A_i A_{i+1} \cdots A_k$ и $A_{k+1} A_{k+2} \cdots A_j$. Это непересекающиеся подцепочки, которые не могут содержать общих матриц. При определении наилучшего способа разрезания стержня длиной n мы ищем наилучшие пути разрезания стержня длиной i при $i = 0, 1, \ldots, n-1$. Поскольку оптимальное решение для длины n включает только одно из решений этих подзадач (после отрезания первой части), независимость подзадач не вызывает никаких сомнений.

Перекрытие подзадач

Вторая составляющая, необходимая для применения динамического программирования, заключается в том, что пространство подзадач должно быть "небольшим" в том смысле, что в результате выполнения рекурсивного алгоритма одни и те же подзадачи решаются снова и снова, а новые подзадачи не возникают. Обычно полное количество различающихся подзадач выражается как полиноми-

14 Зак. 3726

альная функция от объема входных данных. Когда рекурсивный алгоритм снова и снова обращается к одной и той же задаче, говорят, что задача оптимизации содержит ***перекрывающиеся подзадачи*** (overlapping subproblems)[5]. В отличие от описанной выше ситуации, в задачах, решаемых с помощью алгоритма разбиения, на каждом шаге рекурсии обычно возникают полностью новые задачи. В алгоритмах динамического программирования обычно используется преимущество, заключающееся в наличии перекрывающихся подзадач. Это достигается путем однократного решения каждой подзадачи с последующим сохранением результатов в таблице, где при необходимости их можно будет найти за фиксированное время.

В разделе 15.1 мы видели, что рекурсивное решение задачи о разрезании стержня выполняло экспоненциальное количество вызовов для поиска решений меньших подзадач. Наше решение методом динамического программирования превращало экспоненциальный алгоритм в квадратичный.

Чтобы подробнее проиллюстрировать свойство перекрывания подзадач, еще раз обратимся к задаче о перемножении цепочки матриц. Возвратимся к рис. 15.5. Обратите внимание, что процедура MATRIX-CHAIN-ORDER в процессе решения подзадач в более высоких строках постоянно обращается к решениям подзадач в более низких строках. Например, к элементу $m[3,4]$ осуществляется четыре обращения: при вычислении элементов $m[2,4]$, $m[1,4]$, $m[3,5]$ и $m[3,6]$. Если бы элемент $m[3,4]$ каждый раз приходилось вычислять заново, а не просто находить в таблице, это привело бы к значительному увеличению времени работы. Чтобы продемонстрировать это, рассмотрим приведенную ниже (неэффективную) рекурсивную процедуру, в которой определяется величина $m[i,j]$, т.е. минимальное количество скалярных умножений, необходимых для вычисления произведения цепочки матриц $A_{i..j} = A_i A_{i+1} \cdots A_j$. Эта процедура основана непосредственно на рекуррентном соотношении (15.7).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

[5]Может показаться странным, что подзадачи, используемые в динамическом программировании, являются и независимыми, и перекрывающимися. Хотя эти требования и могут показаться противоречащими друг другу, на самом деле это не так, поскольку они относятся к разным понятиям. Две подзадачи одной и той же задачи независимы, если в них не используются общие ресурсы. Две подзадачи перекрываются, если на самом деле речь идет об одной и той же подзадаче, возникающей в разных задачах.

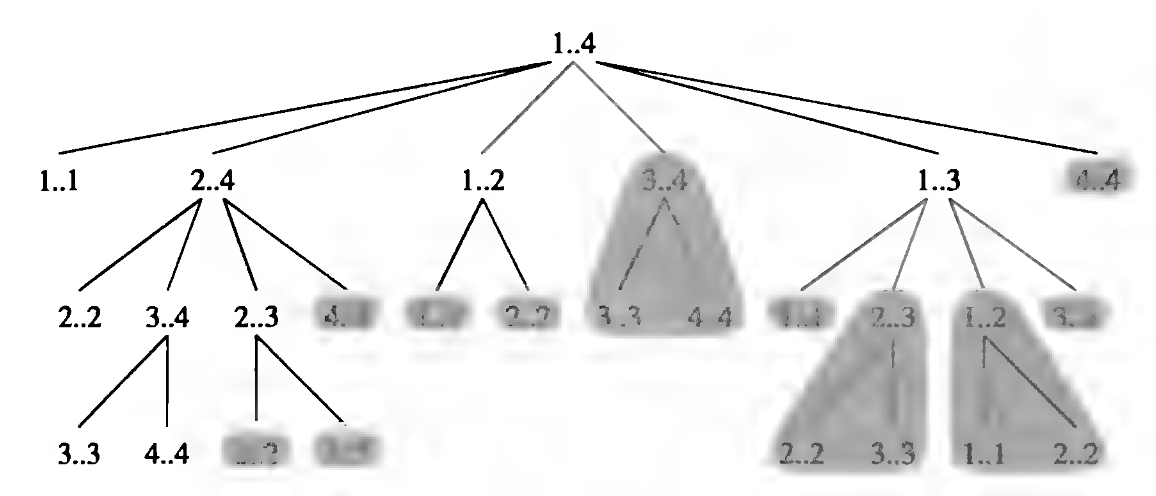


Рис. 15.7. Дерево рекурсии для вычисления RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Каждый узел содержит значения параметров i и j. Вычисления, выполняемые в заштрихованных поддеревьях, заменяются в процедуре MEMOIZED-MATRIX-CHAIN единственным поиском в таблице.

На рис. 15.7 показано дерево рекурсии, полученное в результате вызова процедуры RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Каждый его узел обозначен величинами параметров i и j. Обратите внимание, что некоторые пары значений встречаются много раз.

Можно показать, что время вычисления величины $m[1, n]$ этой рекурсивной процедурой как минимум экспоненциально зависит от n. Обозначим через $T(n)$ время, которое требуется процедуре RECURSIVE-MATRIX-CHAIN для вычисления оптимального способа расстановки скобок в цепочке, состоящей из n матриц. Если считать, что выполнение каждой из строк 1 и 2, 6 и 7 требует как минимум единичного интервала времени, как и умножение в строке 5, то мы получим такое рекуррентное соотношение:

$$T(1) \geq 1 ,$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{для } n > 1 .$$

Если заметить, что при $i = 1, 2, \ldots, n-1$ каждое слагаемое $T(i)$ один раз появляется как $T(k)$ и один раз как $T(n-k)$, и просуммировать $n-1$ единиц с той, которая находится слева от суммы, то рекуррентное соотношение можно переписать в виде

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \qquad (15.8)$$

Докажем с помощью метода подстановок, что $T(n) = \Omega(2^n)$. В частности, покажем, что для всех $n \geq 1$ справедливо соотношение $T(n) \geq 2^{n-1}$. Очевидно, что базисное соотношение индукции выполняется, поскольку $T(1) \geq 1 = 2^0$.

Далее, по методу математической индукции для $n \geq 2$ имеем

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \qquad \text{(согласно (A.5))} \\ &= 2^{n} - 2 + n \\ &\geq 2^{n-1} , \end{aligned}$$

что и завершает доказательство. Таким образом, полное количество вычислений, которые выполняются при вызове процедуры RECURSIVE-MATRIX-CHAIN$(p, 1, n)$, как минимум экспоненциально зависит от n.

Сравним этот нисходящий рекурсивный алгоритм (без запоминания) с восходящим алгоритмом, построенным по принципу динамического программирования. Последний работает эффективнее, поскольку в нем используется свойство перекрывающихся подзадач. Всего в задаче о перемножении цепочки матриц возникает $\Theta(n^2)$ различных подзадач, и в алгоритме, основанном на принципах динамического программирования, каждая из них решается ровно по одному разу. В рекурсивном же алгоритме каждую подзадачу необходимо решать всякий раз, когда она возникает в рекурсивном дереве. Каждый раз, когда рекурсивное дерево для обычного рекурсивного решения задачи несколько раз включает в себя одну и ту же подзадачу, а общее количество подзадач невелико, следует задуматься о возможности применить динамическое программирование, которое может повысить эффективность решения такой задачи, причем зачастую — повысить кардинально.

Построение оптимального решения

На практике мы часто сохраняем в таблице информацию о том, какой выбор был сделан в каждой подзадаче, так что впоследствии нам не нужно дополнительно решать задачу восстановления этой информации.

В задаче перемножения цепочки матриц таблица $s[i, j]$ экономит нам массу работы при построении оптимального решения. Предположим, что мы не поддерживаем таблицу $s[i, j]$, заполняя только лишь таблицу $m[i, j]$ стоимостями оптимальных решений подзадач. В таком случае при построении оптимального решения нам придется осуществлять выбор среди $j - i$ вариантов, чтобы определить, какая из подзадач используется в оптимальном решении задачи о расстановке скобок в цепочке $A_i A_{i+1} \cdots A_j$, а $j - i$ не является константой. Следовательно, время, требуемое для определения выбранной для подзадачи с оптимальным решением, составляет $\Theta(j - i) = \omega(1)$. В случае хранения индекса матрицы для оптимального разбиения цепочки $A_i A_{i+1} \cdots A_j$ каждый выбор восстанавливается за время $O(1)$.

Запоминание

Как мы видели в задаче о разрезании стержня, имеется альтернативный подход динамического программирования, который зачастую обеспечивает эффективность восходящего подхода при применении нисходящей стратегии. Идея заключается в оснащении естественного, но неэффективного алгоритма ***запоминанием***. Как и при восходящем подходе, мы поддерживаем таблицу с решениями подзадач, но управляющая структура для заполнения таблицы больше похожа на рекурсивный алгоритм.

В рекурсивном алгоритме с запоминанием для решения каждой подзадачи поддерживается соответствующий элемент таблицы. Изначально в каждом таком элементе таблицы содержится специальное значение, указывающее на то, что данный элемент еще не заполнен. Если в процессе выполнения рекурсивного алгоритма подзадача встречается впервые, ее решение вычисляется и заносится в таблицу. Каждый раз, когда эта подзадача будет встречаться снова, будет выполняться поиск ее решения в таблице[6].

Ниже приведена версия процедуры RECURSIVE-MATRIX-CHAIN с запоминанием. Обратите внимание на сходство этой процедуры с нисходящим методом с запоминанием, примененным для решения задачи о разрезании стержня.

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length − 1
m[1..n, 1..n] — новая таблица
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

В процедуре MEMOIZED-MATRIX-CHAIN, как и в процедуре MATRIX-CHAIN-ORDER, поддерживается таблица $m[1..n, 1..n]$, состоящая из вычисленных зна-

---

[6] В этом подходе предполагается, что известен набор параметров всех возможных подзадач и установлено соответствие между ячейками таблицы и подзадачами. Другой подход состоит в том, чтобы организовать запоминание с помощью хеширования, в котором в качестве ключей выступают параметры подзадач.

чений $m[i,j]$, которые представляют собой минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Изначально каждый элемент таблицы содержит значение ∞, указывающее на то, что данный элемент еще должен быть заполнен. Если при вызове процедуры LOOKUP-CHAIN(m,p,i,j) в строке 1 выполняется условие $m[i,j] < \infty$, то эта процедура в строке 2 просто возвращает ранее вычисленную стоимость $m[i,j]$. В противном случае эта стоимость вычисляется так же, как и в процедуре RECURSIVE-MATRIX-CHAIN, сохраняется в элементе $m[i,j]$, после чего возвращается. Таким образом, процедура LOOKUP-CHAIN(m,p,i,j) всегда возвращает значение $m[i,j]$, но оно вычисляется лишь в том случае, если это первый вызов данной процедуры с этими параметрами i и j.

На рис. 15.7 показано, насколько эффективнее процедура MEMOIZED-MATRIX-CHAIN расходует время по сравнению с процедурой RECURSIVE-MATRIX-CHAIN. Затененные поддеревья представляют значения, которые вычисляются только один раз. При возникновении повторной потребности в этих значениях вместо их вычисления осуществляется поиск в хранилище.

Время работы процедуры MEMOIZED-MATRIX-CHAIN, как и время выполнения алгоритма MATRIX-CHAIN-ORDER, построенного по принципу динамического программирования, равно $O(n^3)$. Строка 5 процедуры MEMOIZED-MATRIX-CHAIN выполняется $\Theta(n^2)$ раз. Все вызовы процедуры LOOKUP-CHAIN можно разбить на два типа:

1. вызовы, в которых выполняется условие $m[i,j] = \infty$, так что выполняются строки 3–9;
2. вызовы, в которых выполняется условие $m[i,j] < \infty$, так что процедура LOOKUP-CHAIN просто выполняет возврат в строке 2.

Всего вызовов первого типа насчитывается $\Theta(n^2)$, по одному на каждый элемент таблицы. Все вызовы второго типа осуществляются как рекурсивные обращения из вызовов первого типа. Всякий раз, когда в некотором вызове процедуры LOOKUP-CHAIN выполняются рекурсивные обращения, общее их количество равно $O(n)$. Поэтому в общем итоге производится $O(n^3)$ вызовов второго типа, причем на каждый из них расходуется время $O(1)$. На каждый вызов первого типа требуется время $O(n)$ плюс время, затраченное на рекурсивные обращения в данном вызове первого типа. Поэтому общее время равно $O(n^3)$. Таким образом, запоминание преобразует алгоритм, время работы которого равно $\Omega(2^n)$, в алгоритм со временем работы $O(n^3)$.

В итоге получается, что задачу об оптимальном способе перемножения цепочки матриц можно решить либо с помощью алгоритма с запоминанием, работающего в нисходящем направлении, либо с помощью динамического программирования в восходящем направлении. При этом в обоих случаях потребуется время, равное $O(n^3)$. Оба метода используют преимущество, возникающее благодаря перекрытию подзадач. Всего возникает только $\Theta(n^2)$ различных подзадач, и в каждом из описанных выше методов решение каждой из них вычисляется один раз. Если не применять запоминание, то время работы обычного рекур-

сивного алгоритма станет экспоненциальным, поскольку уже решенные задачи придется неоднократно решать заново.

В общем случае, если все подзадачи необходимо решить хотя бы по одному разу, восходящий алгоритм, построенный по принципу динамического программирования, обычно работает быстрее, чем нисходящий алгоритм с запоминанием. Причины — отсутствие непроизводительных накладных расходов на рекурсию и их сокращение при поддержке таблицы. Более того, в некоторых задачах после определенных исследований удается сократить время доступа к таблице или занимаемый ею объем. Если же можно обойтись без решения некоторых подзадач, содержащихся в пространстве подзадач данной задачи, запоминание результатов решений обладает тем преимуществом, что решаются только действительно необходимые подзадачи.

Упражнения

15.3.1

Как эффективнее определить оптимальное количество скалярных умножений в задаче о перемножении цепочки матриц: перечислить все способы расстановки скобок в произведении матриц и вычислить количество скалярных умножений в каждом из них или запустить процедуру RECURSIVE-MATRIX-CHAIN? Обоснуйте ответ.

15.3.2

Изобразите рекурсивное дерево для процедуры MERGE-SORT, описанной в разделе 2.3.1 и работающей с массивом из 16 элементов. Объясните, почему для повышения производительности работы хорошего алгоритма “разделяй и властвуй”, такого как MERGE-SORT, использование запоминания не даст никакого улучшения производительности.

15.3.3

Рассмотрим разновидность задачи о перемножении цепочки матриц, цель которой — так расставить скобки в последовательности матриц, чтобы количество скалярных умножений стало не минимальным, а максимальным. Проявляется ли в этой задаче оптимальная подструктура?

15.3.4

Как говорилось выше, в динамическом программировании сначала решаются подзадачи, а затем выбираются те из них, которые будут использованы в оптимальном решении задачи. Профессор утверждает, что не всегда необходимо решать все подзадачи, чтобы найти оптимальное решение. Он выдвигает предположение о том, что оптимальное решение задачи о перемножении цепочки матриц можно найти, всегда выбирая матрицу A_k, после которой следует разбивать произведение $A_i A_{i+1} \cdots A_j$ (индекс k выбирается так, чтобы минимизировать величину $p_{i-1} p_k p_j$) *перед* решением подзадач. Приведите пример задачи о перемножении цепочки матриц, в котором такой жадный подход приводит к решению, отличному от оптимального.

15.3.5

Предположим, что в задаче о разрезании стержня из раздела 15.1 имеется предельное значение l_i количества частей длиной i ($i = 1, 2, \ldots, n$), которое можно получить при разрезании одного стержня. Покажите, что свойство оптимальной подструктуры из раздела 15.1 в этом случае не выполняется.

15.3.6

Представим, что вы хотите обменять одну валюту на другую. Вы надеетесь на то, что если вместо непосредственного обмена выполнить несколько промежуточных обменов на другие валюты, то обмен получится более выгодным. Предположим, что можно выполнять обмен n различных валют, пронумерованных $1, 2, \ldots, n$, причем вы начинаете с валюты 1 и хотите закончить валютой n. Для каждой пары валют i и j у вас имеется обменный курс r_{ij}, означающий, что если у вас есть d единиц валюты i, то вы можете получить за них dr_{ij} единиц валюты j. Последовательность обменов может вызывать комиссионные сборы, зависящие от количества произведенных обменов. Пусть c_k — комиссионный сбор, который вы должны заплатить за k обменов. Покажите, что, если $c_k = 0$ для всех $k = 1, 2, \ldots, n$, то задача поиска наилучшей последовательности обменов валюты 1 на валюту n демонстрирует оптимальную подструктуру. Затем покажите, что если комиссионные сборы c_k имеют произвольные значения, то эта задача поиска наилучшей последовательности обменов валюты 1 на валюту n такой оптимальной подструктурой может и не обладать.

15.4. Наидлиннейшая общая подпоследовательность

В биологических приложениях часто возникает необходимость сравнить ДНК двух (или более) различных организмов. Стандартная ДНК состоит из последовательности молекул, которые называются ***основаниями*** (bases). К этим молекулам относятся: аденин (adenine), гуанин (guanine), цитозин (cytosine) и тимин (thymine). Если обозначить каждое из перечисленных оснований его начальной буквой латинского алфавита, то стандартную ДНК можно представить в виде строки, состоящей из конечного множества элементов $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. (Определение строки см. в приложении В.) Например, ДНК одного организма может иметь вид S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, а ДНК другого — S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Одна из целей сравнения двух ДНК состоит в том, чтобы выяснить степень их сходства. Это один из показателей того, насколько тесно связаны между собой два организма. Степень подобия можно определить многими способами. Например, можно сказать, что два кода ДНК подобны, если один из них является подстрокой другого. (Алгоритмы, с помощью которых решается эта задача, исследуются в главе 32.) В нашем примере ни S_1, ни S_2 не является подстрокой другой ДНК. В этом случае можно сказать, что две цепочки молекул подобны, если для преобразования одной из них в другую потребовались бы только небольшие изменения. (Такой подход рассматривается

в задаче 15.5.) Еще один способ определения степени подобия последовательностей S_1 и S_2 заключается в поиске третьей последовательности S_3, основания которой имеются как в S_1, так и в S_2; при этом они следуют в одном и том же порядке, но не обязательно одно за другим. Чем длиннее последовательность S_3, тем более схожи последовательности S_1 и S_2. В рассматриваемом примере наидлиннейшая последовательность S_3 имеет вид GTCGTCGGAAGCCGGCCGAA.

Последнее из упомянутых выше понятий подобия формализуется в виде задачи о наидлиннейшей общей подпоследовательности. Подпоследовательность данной последовательности — это просто данная последовательность, из которой удалили нуль или более элементов. Формально последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ является ***подпоследовательностью*** (subsequence) последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$, если существует строго возрастающая последовательность $\langle i_1, i_2, \ldots, i_k \rangle$ индексов X, такая, что для всех $j = 1, 2, \ldots, k$ выполняется соотношение $x_{i_j} = z_j$. Например, $Z = \langle B, C, D, B \rangle$ представляет собой подпоследовательность последовательности $X = \langle A, B, C, B, D, A, B \rangle$, причем соответствующая ей последовательность индексов имеет вид $\langle 2, 3, 5, 7 \rangle$.

Говорят, что последовательность Z является ***общей подпоследовательностью*** (common subsequence) последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Например, если $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$, то последовательность $\langle B, C, A \rangle$ является общей подпоследовательностью X и Y. Однако последовательность $\langle B, C, A \rangle$ — не *наидлиннейшая* общая подпоследовательность X и Y, поскольку ее длина равна 3, а длина последовательности $\langle B, C, B, A \rangle$, которая также является общей подпоследовательностью X и Y, равна 4. Последовательность $\langle B, C, B, A \rangle$ — *наидлиннейшая общая подпоследовательность* (longest common subsequence — LCS) последовательностей X и Y, как и последовательность $\langle B, D, A, B \rangle$, поскольку не существует общей подпоследовательности длиной 5 элементов или более.

В ***задаче о наидлиннейшей общей подпоследовательности*** задаются две последовательности, $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$, и требуется найти общую подпоследовательность X и Y максимальной длины. В данном разделе показано, что эта задача эффективно решается методом динамического программирования.

Этап 1. Характеристика наидлиннейшей общей подпоследовательности

Решение задачи поиска наидлиннейшей общей подпоследовательности "в лоб" заключается в том, чтобы перечислить все подпоследовательности последовательности X и проверить, являются ли они также подпоследовательностями Y, пытаясь отыскать при этом наидлиннейшую из них. Каждая подпоследовательность последовательности X соответствует подмножеству индексов $\{1, 2, \ldots, m\}$ последовательности X. Всего имеется 2^m подпоследовательностей последовательности X, поэтому время работы алгоритма, основанного на этом подходе, будет экспоненциально зависеть от размера задачи, и для длинных последовательностей он становится непригодным.

Однако задача поиска наидлиннейшей общей подпоследовательности обладает оптимальной подструктурой. Этот факт доказывается в сформулированной ниже

теореме. Как мы увидим, естественно возникающие классы подзадач соответствуют парам "префиксов" двух входных последовательностей. Дадим точное определение этого понятия: i-м ***префиксом*** последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ для $i = 1, 2, \ldots, m$ является подпоследовательность $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. Например, если $X = \langle A, B, C, B, D, A, B \rangle$, то $X_4 = \langle A, B, C, B \rangle$, а X_0 представляет собой пустую последовательность.

Теорема 15.1 (Оптимальная подструктура LCS)

Пусть имеются последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ и пусть $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — наидлиннейшая общая подпоследовательность X и Y.

1. Если $x_m = y_n$, то $z_k = x_m = y_n$ и Z_{k-1} — LCS последовательностей X_{m-1} и Y_{n-1}.
2. Если $x_m \neq y_n$, то из $z_k \neq x_m$ вытекает, что Z представляет собой LCS последовательностей X_{m-1} и Y.
3. Если $x_m \neq y_n$, то из $z_k \neq y_n$ вытекает, что Z представляет собой LCS последовательностей X и Y_{n-1}.

Доказательство. (1) Если бы выполнялось соотношение $z_k \neq x_m$, то к последовательности Z можно было бы добавить элемент $x_m = y_n$, в результате чего получилась бы общая подпоследовательность последовательностей X и Y длиной $k + 1$, а это противоречит предположению о том, что Z — *наидлиннейшая* общая подпоследовательность последовательностей X и Y. Таким образом, должно выполняться соотношение $z_k = x_m = y_n$. Итак, префикс Z_{k-1} — общая подпоследовательность последовательностей X_{m-1} и Y_{n-1} длиной $k - 1$. Нужно показать, что это наидлиннейшая общая подпоследовательность. Проведем доказательство методом от противного. Предположим, что имеется общая подпоследовательность W последовательностей X_{m-1} и Y_{n-1}, длина которой превышает $k - 1$. Добавив к W элемент $x_m = y_n$, получим общую подпоследовательность последовательностей X и Y, длина которой превышает k, что приводит к противоречию.

(2) Если $z_k \neq x_m$, то Z — общая подпоследовательность последовательностей X_{m-1} и Y. Если бы существовала общая подпоследовательность W последовательностей X_{m-1} и Y, длина которой превышала бы k, то она была бы также общей подпоследовательностью последовательностей X_m и Y, что противоречит предположению о том, что Z — наидлиннейшая общая подпоследовательность последовательностей X и Y.

(3) Доказательство симметрично (2). ■

Из теоремы 15.1 видно, что наидлиннейшая общая подпоследовательность двух последовательностей содержит в себе наидлиннейшую общую подпоследовательность их префиксов. Таким образом, задача о наидлиннейшей общей подпоследовательности обладает оптимальной подструктурой. В рекурсивном решении этой задачи также возникают перекрывающиеся подзадачи, но об этом речь пойдет чуть позже.

Этап 2. Рекурсивное решение

Из теоремы 15.1 следует, что при нахождении наидлиннейшей общей подпоследовательности последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ возникает одна или две подзадачи. Если $x_m = y_n$, необходимо найти наидлиннейшую общую подпоследовательность последовательностей X_{m-1} и Y_{n-1}. Добавив к ней элемент $x_m = y_n$, получим LCS последовательностей X и Y. Если $x_m \neq y_n$, необходимо решить две подзадачи: найти LCS последовательностей X_{m-1} и Y, а также последовательностей X и Y_{n-1}. Какая из этих двух подпоследовательностей окажется длиннее, та и будет наидлиннейшей общей подпоследовательностью последовательностей X и Y. Поскольку эти случаи исчерпывают все возможности, мы знаем, что одно из оптимальных решений подзадач должно находиться в LCS X и Y.

В задаче поиска наидлиннейшей общей подпоследовательности легко увидеть проявление свойства перекрывающихся подзадач. Чтобы найти LCS последовательностей X и Y, может понадобиться найти LCS последовательностей X и Y_{n-1}, а также последовательностей X_{m-1} и Y. Однако в каждой из этих подзадач возникает подзадача поиска LCS последовательностей X_{m-1} и Y_{n-1}. Общие подзадачи возникают и во многих других случаях.

Как и в задаче о перемножении цепочки матриц, в рекурсивном решении задачи о наидлиннейшей общей подпоследовательности устанавливается рекуррентное соотношение для значений, характеризующих оптимальное решение. Обозначим через $c[i, j]$ длину наидлиннейшей общей подпоследовательности последовательностей X_i и Y_j. Если $i = 0$ или $j = 0$, длина одной из этих последовательностей равна нулю, поэтому наидлиннейшая их общая подпоследовательность имеет нулевую длину. Оптимальная вспомогательная подструктура задачи о наидлиннейшей общей подпоследовательности определяется рекурсивной формулой

$$c[i,j] = \begin{cases} 0\,, & \text{если } i = 0 \text{ или } j = 0\,, \\ c[i-1, j-1] + 1\,, & \text{если } i, j > 0 \text{ и } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j])\,, & \text{если } i, j > 0 \text{ и } x_i \neq y_j\,. \end{cases} \quad (15.9)$$

Обратите внимание, что в этой рекурсивной формулировке условия задачи ограничивают круг рассматриваемых подзадач. Если $x_i = y_j$, можно и нужно рассматривать подзадачу поиска наидлиннейшей общей подпоследовательности последовательностей X_{i-1} и Y_{j-1}. В противном случае рассматриваются две подзадачи поиска LCS последовательностей X_i и Y_{j-1}, а также X_{i-1} и Y_j. В рассмотренных ранее алгоритмах, основанных на принципах динамического программирования (для задач о разрезании стержня и о перемножении цепочки матриц), выбор подзадач не зависел от условий задачи. Задача поиска LCS не единственная, в которой вид возникающих подзадач определяется условиями задачи. В качестве другого подобного примера можно привести задачу о расстоянии редактирования (см. задачу 15.5).

Этап 3. Вычисление длины наидлиннейшей общей подпоследовательности

На основе уравнения (15.9) легко написать рекурсивный алгоритм с экспоненциальным временем работы, предназначенный для вычисления длины LCS двух последовательностей. Однако благодаря наличию всего лишь $\Theta(mn)$ различных подзадач можно воспользоваться динамическим программированием для вычисления решения в восходящем направлении.

В процедуре LCS-LENGTH в роли входных данных выступают две последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Величины $c[i, j]$ хранятся в таблице $c[0..m, 0..n]$, элементы которой вычисляются построчно (т.е. сначала слева направо заполняется первая строка, затем — вторая и т.д.). В процедуре также поддерживается таблица $b[1..m, 1..n]$, благодаря которой упрощается процесс построения оптимального решения. Наглядный смысл элементов $b[i, j]$ состоит в том, что каждый из них указывает на элемент таблицы, соответствующий оптимальному решению подзадачи, выбранной при вычислении элемента $c[i, j]$. Процедура возвращает таблицы b и c; в элементе $c[m, n]$ хранится длина LCS последовательностей X и Y.

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
b[1..m, 1..n] и c[0..m, 0..n] — новые таблицы
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c и b
```

На рис. 15.8 показаны таблицы, полученные в результате работы процедуры LCS-LENGTH с последовательностями $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$. Время работы процедуры равно $\Theta(mn)$, поскольку каждый элемент таблицы вычисляется за время $\Theta(1)$.

Этап 4. Построение наидлиннейшей общей подпоследовательности

С помощью таблицы b, которая возвращается процедурой LCS-LENGTH, можно быстро построить самую длинную общую подпоследовательность последо-

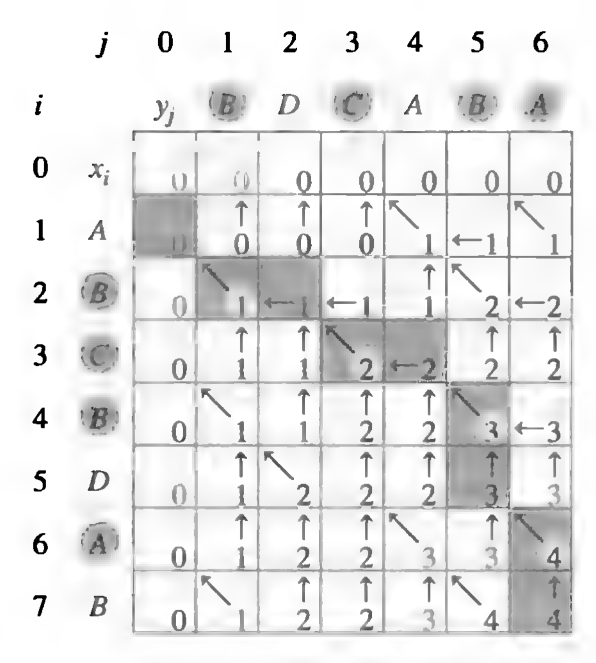


Рис. 15.8. Таблицы c и b, полученные в результате работы процедуры LCS-LENGTH с последовательностями $X = \langle A, B, C, B, D, A, B\rangle$ и $Y = \langle B, D, C, A, B, A\rangle$. Квадрат на пересечении строки i и столбца j содержит значение $c[i, j]$ и стрелку в качестве значения элемента $b[i, j]$. Значение 4 в квадрате $c[7, 6]$ — нижнем правом углу таблицы — представляет собой длину LCS $\langle B, C, B, A\rangle$ последовательностей X и Y. При $i, j > 0$ элемент $c[i, j]$ зависит только от того, выполняется ли равенство $x_i = y_j$, и от значений в элементах $c[i-1, j]$, $c[i, j-1]$ и $c[i-1, j-1]$, которые вычисляются до вычисления $c[i, j]$. Чтобы восстановить элементы, входящие в LCS, следуем по стрелкам $b[i, j]$ из нижнего правого угла таблицы; на рисунке эта последовательность заштрихована. Каждая стрелка "$\nwarrow$" в этой последовательности соответствует элементу, для которого $x_i = y_j$ является членом LCS.

вательностей $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$. Мы просто начинаем с элемента $b[m, n]$ и проходим таблицу по стрелкам. Если значением элемента $b[i, j]$ является "$\nwarrow$", то элемент $x_i = y_j$ принадлежит наидлиннейшей общей подпоследовательности. Элементы LCS, восстановленные этим методом, следуют в обратном порядке. Приведенная ниже рекурсивная процедура выводит элементы самой длинной общей подпоследовательности последовательностей X и Y в прямом порядке. Начальный вызов этой процедуры выглядит как PRINT-LCS$(b, X, X.length, Y.length)$.

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Для таблицы b, изображенной на рис. 15.8, эта процедура выводит последовательность "$BCBA$". Время работы процедуры равно $O(m + n)$, поскольку хотя бы один из индексов i или j уменьшается при каждом рекурсивном вызове.

Улучшение кода

После разработки алгоритма часто оказывается, что можно улучшить время его работы или объем требуемой им памяти. Одни изменения могут упростить код и уменьшить постоянный множитель, но при этом не приводят к повышению производительности в асимптотическом пределе. Другие же могут вызвать существенную асимптотическую экономию времени работы и используемой памяти.

Например, в алгоритме поиска LCS можно обойтись без таблицы b. Каждый элемент $c[i,j]$ зависит только от трех других элементов этой же таблицы: $c[i-1,j-1]$, $c[i-1,j]$ и $c[i,j-1]$. Для данного элемента $c[i,j]$ в течение времени $O(1)$ можно определить, какое из этих трех значений было использовано для вычисления $c[i,j]$, не прибегая к таблице b. Таким образом, самую длинную общую подпоследовательность можно восстановить в течение времени $O(m+n)$. Для этого понадобится процедура, подобная процедуре PRINT-LCS (такую процедуру предлагается составить в упр. 15.4.2). Несмотря на то что в этом методе экономится объем памяти, равный $\Theta(mn)$, асимптотически количество памяти, необходимой для вычисления самой длинной общей подпоследовательности, не изменяется. Это объясняется тем, что таблица c все равно требует $\Theta(mn)$ памяти.

Однако можно уменьшить асимптотический объем памяти, необходимой для работы процедуры PRINT-LCS, поскольку одновременно нужны лишь две строки таблицы c: вычисляемая и предыдущая. (Фактически для вычисления длины самой длинной общей подпоследовательности можно обойтись пространством, лишь немного превышающим объем, необходимый для одной строки матрицы c; см. упр. 15.4.4.) Это усовершенствование работает лишь в том случае, когда нужно знать только длину самой длинной общей подпоследовательности. Если же необходимо воссоздать элементы этой подпоследовательности, такая "урезанная" таблица содержит недостаточно информации для того, чтобы проследить обратные шаги за время $O(m+n)$.

Упражнения

15.4.1

Определите LCS последовательностей $\langle 1,0,0,1,0,1,0,1 \rangle$ и $\langle 0,1,0,1,1,0,1,1,0 \rangle$.

15.4.2

Напишите псевдокод для воссоздания наидлиннейшей общей подпоследовательности из заполненной таблицы c и исходных последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ за время $O(m+n)$, не используя таблицу b.

15.4.3

Приведите версию процедуры LCS-LENGTH с запоминанием, время работы которой равно $O(mn)$.

15.4.4

Покажите, как вычислить длину наидлиннейшей общей подпоследовательности, используя при этом только $2 \cdot \min(m,n)$ элементов таблицы c плюс $O(1)$ допол-

нительной памяти. Затем покажите, как то же самое можно сделать с помощью $\min(m, n)$ элементов таблицы и $O(1)$ дополнительной памяти.

15.4.5

Разработайте алгоритм, предназначенный для вычисления наидлиннейшей монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O(n^2)$.

15.4.6 ★

Разработайте алгоритм, предназначенный для вычисления наидлиннейшей монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O(n \lg n)$. (*Указание:* обратите внимание, что последний элемент кандидата в искомую подпоследовательность длиной i по величине не меньше последнего элемента кандидата длиной $i - 1$.)

15.5. Оптимальные бинарные деревья поиска

Предположим, что разрабатывается программа, предназначенная для перевода текстов с русского языка на украинский. Для каждого русского слова необходимо найти украинский эквивалент. Один из возможных путей поиска — построение бинарного дерева поиска с n русскими словами, выступающими в роли ключей, и украинскими эквивалентами, играющими роль сопутствующих данных. Поскольку поиск с помощью этого дерева будет производиться для каждого отдельного слова из текста, полное затраченное на него время должно быть как можно меньше. С помощью красно-черного дерева или любого другого сбалансированного бинарного дерева поиска можно добиться того, что время каждого отдельного поиска будет равным $O(\lg n)$. Однако слова встречаются с разной частотой, и может получиться так, что какое-нибудь часто употребляемое слово (например, предлог или союз) находится далеко от корня, а такое редкое слово, как *контрвстреча*, — возле корня. Подобный способ организации привел бы к замедлению перевода, поскольку количество узлов, просмотренных в процессе поиска ключа в бинарном дереве, равно увеличенной на единицу глубине узла, содержащего данный ключ. Нужно сделать так, чтобы слова, которые встречаются в тексте часто, были размещены поближе к корню.[7] Кроме того, в исходном тексте могут встречаться слова, для которых перевод отсутствует. Таких слов может вообще не оказаться в бинарном дереве поиска. Как организовать бинарное дерево поиска, чтобы свести к минимуму количество посещенных в процессе поиска узлов, если известно, с какой частотой встречаются слова?

То, что мы хотим получить, называется ***оптимальное бинарное дерево поиска*** (optimal binary search tree). Приведем формальное описание задачи. Имеется

[7]Впрочем, одни и те же слова в текстах разной тематики могут встречаться с разной частотой.

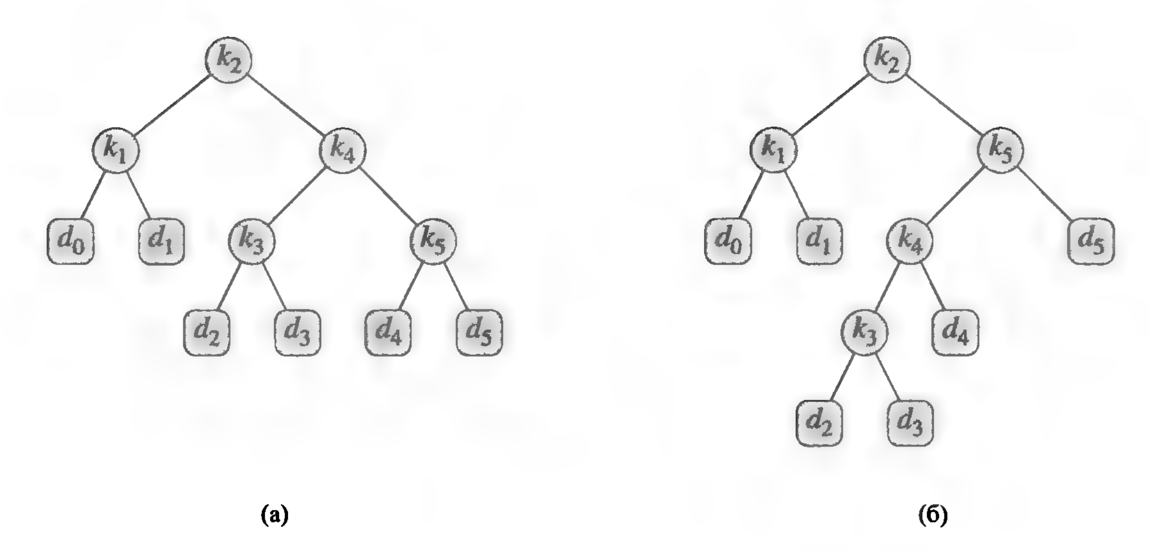


Рис. 15.9. Два бинарных дерева поиска для множества из $n = 5$ ключей со следующими вероятностями.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(а) Бинарное дерево поиска с ожидаемой стоимостью поиска 2.80. **(б)** Бинарное дерево поиска с ожидаемой стоимостью поиска 2.75. Это дерево — оптимальное.

заданная последовательность $K = \langle k_1, k_2, \ldots, k_n \rangle$, состоящая из n различных ключей, которые расположены в отсортированном порядке (так что $k_1 < k_2 < \cdots < k_n$). Из этих ключей нужно составить бинарное дерево поиска. Для каждого ключа k_i задана вероятность p_i поиска этого ключа. Кроме того, может выполняться поиск значений, отсутствующих в последовательности K, поэтому следует предусмотреть $n+1$ фиктивных ключей $d_0, d_1, d_2, \ldots, d_n$, представляющих эти значения. В частности, d_0 представляет все значения, меньшие k_1, а d_n — все значения, превышающие k_n. Фиктивный ключ d_i ($i = 1, 2, \ldots, n-1$) представляет все значения, которые находятся между k_i и k_{i+1}. Для каждого фиктивного ключа d_i задана соответствующая ей вероятность q_i. На рис. 15.9 показаны два бинарных дерева поиска для множества, состоящего из $n = 5$ ключей. Каждый ключ k_i представлен внутренним узлом, а каждый фиктивный ключ d_i является листом. Поиск может быть либо успешным (найден какой-то ключ k_i), либо неудачным (возвращается какой-то фиктивный ключ d_i), поэтому справедливо соотношение

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{15.10}$$

Поскольку вероятность поиска каждого обычного и фиктивного ключа известна, можно определить математическое ожидание стоимости поиска по заданному бинарному дереву поиска T. Предположим, что фактическая стоимость поиска определяется количеством проверенных узлов, т.е. увеличенной на единицу глубиной узла на дереве T, в котором находится искомый ключ. Тогда математиче-

ское ожидание стоимости поиска в дереве T равно

$$\begin{aligned} \text{E}[\text{стоимость поиска в } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \ , \end{aligned} \quad (15.11)$$

где величина depth_T обозначает глубину узла в дереве T. Последнее равенство следует из уравнения (15.10). Используя приведенную на рис. 15.9 таблицу вероятностей, можно вычислить математическое ожидание стоимости поиска для бинарного дерева, изображенного в части (а) рисунка.

Узел	Глубина	Вероятность	Вклад
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Всего			2.80

Наша цель — построить для данного набора вероятностей бинарное дерево поиска, математическое ожидание стоимости поиска для которого будет минимальным. Такое дерево называется ***оптимальным бинарным деревом поиска***. На рис. 15.9, (б) показано оптимальное бинарное дерево поиска для вероятностей, указанных в таблице в подписи к рисунку. Математическое ожидание стоимости поиска в этом дереве равно 2.75. Этот пример демонстрирует, что оптимальное бинарное дерево поиска — это не обязательно дерево минимальной высоты. Кроме того, в оптимальном дереве ключ, которому соответствует максимальная вероятность, не всегда находится в корне. В данном случае вероятность имеет самую большую величину для ключа k_5, хотя в корне оптимального бинарного дерева расположен ключ k_2. (Минимальная величина математического ожидания стоимости поиска для всевозможных бинарных деревьев поиска, в корне которых находится ключ k_5, равна 2.85.)

Как и в задаче о перемножении цепочки матриц, последовательный перебор всех возможных деревьев в данном случае оказывается неэффективным. Чтобы сконструировать бинарное дерево поиска, можно обозначить ключами $k_1, k_2, \ldots, k_n$ узлы бинарного дерева с n узлами, а затем добавить листья для фиктивных ключей. В задаче 12.4 было показано, что количество бинарных деревьев с n узлами равно $\Omega(4^n/n^{3/2})$, так что количество бинарных деревьев, которые нужно проверять при полном переборе, растет экспоненциально с ростом n. Не

удивительно, что эта задача будет решаться методом динамического программирования.

Этап 1. Структура оптимального бинарного дерева поиска

Чтобы охарактеризовать оптимальную подструктуру оптимального бинарного дерева поиска, исследуем его поддеревья. Рассмотрим произвольное поддерево бинарного дерева поиска. Оно должно содержать ключи в непрерывном интервале $k_i, \ldots, k_j$ для некоторых $1 \leq i \leq j \leq n$. Кроме того, поддерево, содержащее ключи $k_i, \ldots, k_j$, должно также содержать в качестве листьев фиктивные ключи $d_{i-1}, \ldots, d_j$.

Теперь можно сформулировать оптимальную подструктуру: если в состав оптимального бинарного дерева поиска T входит поддерево T', содержащее ключи $k_i, \ldots, k_j$, это поддерево также должно быть оптимальным для подзадачи с ключами $k_i, \ldots, k_j$ и фиктивными ключами $d_{i-1}, \ldots, d_j$. Для доказательства этого утверждения применяется обычный метод вырезания и вставки. Если бы существовало поддерево T'', математическое ожидание поиска в котором ниже, чем математическое ожидание поиска в поддереве T', то из дерева T можно было бы вырезать поддерево T' и подставить вместо него поддерево T''. В результате получилось бы дерево, математическое ожидание времени поиска в котором оказалось бы меньше, что противоречит утверждению об оптимальности дерева T.

Покажем с помощью описанной выше оптимальной подструктуры, что оптимальное решение задачи можно воссоздать из оптимальных решений подзадач. Если имеется поддерево, содержащее ключи $k_i, \ldots, k_j$, то один из этих ключей, скажем, k_r ($i \leq r \leq j$) будет корнем этого оптимального поддерева. Поддерево, которое находится слева от корня k_r, будет содержать ключи $k_i, \ldots, k_{r-1}$ (и фиктивные ключи $d_{i-1}, \ldots, d_{r-1}$), а правое поддерево — ключи $k_{r+1}, \ldots, k_j$ (и фиктивные ключи $d_r, \ldots, d_j$). Как только будут проверены все ключи k_r (где $i \leq r \leq j$), которые являются кандидатами на роль корня, и найдены оптимальные бинарные деревья поиска, содержащие элементы $k_i, \ldots, k_{r-1}$ и $k_{r+1}, \ldots, k_j$, мы гарантированно построим оптимальное бинарное дерево поиска.

Стоит сделать одно замечание по поводу "пустых" поддеревьев. Предположим, что в поддереве с ключами $k_i, \ldots, k_j$ в качестве корня выбран ключ k_i. Согласно приведенным выше рассуждениям, поддерево, которое находится слева от корня k_i, содержит ключи $k_i, \ldots, k_{i-1}$. Естественно интерпретировать эту последовательность как такую, в которой не содержится ни одного ключа. Однако следует иметь в виду, что поддеревья содержат, помимо реальных, и фиктивные ключи. Примем соглашение, согласно которому поддерево, состоящее из ключей $k_i, \ldots, k_{i-1}$, не содержит обычных ключей, но содержит один фиктивный ключ d_{i-1}. Аналогично, если в качестве корня выбран ключ k_j, то правое поддерево не содержит обычных ключей, но содержит один фиктивный ключ d_j.

Этап 2. Рекурсивное решение

Теперь все готово для рекурсивного определения оптимального решения. В качестве подзадачи выберем задачу поиска оптимального бинарного дерева поиска, содержащего ключи $k_i, \ldots, k_j$, где $i \geq 1$, $j \leq n$ и $j \geq i-1$ (если $j = i-1$, то факти-

ческих ключей не существует, имеется только фиктивный ключ d_{i-1}). Определим величину $e[i,j]$ как математическое ожидание стоимости поиска в оптимальном бинарном дереве поиска с ключами $k_i,\ldots,k_j$. В конечном итоге нужно вычислить величину $e[1,n]$.

Если $j = i - 1$, то все просто. В этом случае имеется всего один фиктивный ключ d_{i-1}, и математическое ожидание стоимости поиска равно $e[i,i-1] = q_{i-1}$.

Если $j \geq i$, то среди ключей $k_i,\ldots,k_j$ нужно выбрать корень k_r, а затем из ключей $k_i,\ldots,k_{r-1}$ составить левое оптимальное бинарное дерево поиска, а из ключей $k_{r+1},\ldots,k_j$ — правое оптимальное бинарное дерево поиска. Что происходит с математическим ожиданием стоимости поиска в поддереве, когда оно становится поддеревом какого-то узла? Глубина каждого узла в поддереве возрастает на единицу. Согласно уравнению (15.11) математическое ожидание стоимости поиска в этом поддереве возрастает на величину суммы по всем вероятностям поддерева. Обозначим эту сумму вероятностей, вычисленную для поддерева с ключами $k_i,\ldots,k_j$, как

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l . \qquad (15.12)$$

Таким образом, если k_r — корень оптимального поддерева, содержащего ключи $k_i,\ldots,k_j$, то выполняется соотношение

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) .$$

Заметив, что

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j) ,$$

перепишем $e[i,j]$ как

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) . \qquad (15.13)$$

Рекурсивное соотношение (15.13) предполагает, что нам известно, какой узел k_r используется в качестве корня. На эту роль выбирается ключ, который приводит к минимальному значению математического ожидания стоимости поиска. С учетом этого получаем окончательную рекурсивную формулу:

$$e[i,j] = \begin{cases} q_{i-1} , & \text{если } j = i-1 , \\ \min\limits_{i \leq r \leq j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} , & \text{если } i \leq j . \end{cases} \qquad (15.14)$$

Величины $e[i,j]$ представляют собой математическое ожидание стоимостей поиска в оптимальных бинарных деревьях поиска. Чтобы было легче следить за структурой оптимального бинарного дерева поиска, обозначим через $root[i,j]$ (где $1 \leq i \leq j \leq n$) индекс r узла k_r, который является корнем оптимального бинарного дерева поиска, содержащего ключи $k_i,\ldots,k_j$. Хотя вскоре мы узнаем, как вычисляются величины $root[i,j]$, способ восстановления из этих величин оптимального бинарного дерева поиска отложим до упр. 15.5.1.

Этап 3. Вычисление математического ожидания стоимости поиска в оптимальном бинарном дереве поиска

На данном этапе можно заметить некоторое сходство между характеристиками задач об оптимальных бинарных деревьях поиска и о перемножении цепочки матриц. Во вспомогательных задачах обеих задач индексы элементов изменяются последовательно. Прямая рекурсивная реализация уравнения (15.14) может оказаться такой же неэффективной, как и прямая рекурсивная реализация алгоритма в задаче о перемножении цепочки матриц. Вместо этого будем сохранять значения $e[i,j]$ в таблице $e[1..n+1, 0..n]$. Первый индекс должен пробегать не n, а $n+1$ значений. Это объясняется тем, что для получения поддерева, в которое входит только фиктивный ключ d_n, понадобится вычислить и сохранить значение $e[n+1,n]$. Второй индекс должен начинаться с нуля, поскольку для получения поддерева, содержащего лишь фиктивный ключ d_0, нужно вычислить и сохранить значение $e[1,0]$. Мы будем использовать только те элементы $e[i,j]$, для которых $j \geq i-1$. Кроме того, будет использована таблица $root[i,j]$, в которую будут заноситься корни поддеревьев, содержащих ключи $k_i, \ldots, k_j$. В этой таблице задействованы только те записи, для которых $1 \leq i \leq j \leq n$.

Для повышения эффективности понадобится еще одна таблица. Вместо того чтобы каждый раз при вычислении $e[i,j]$ вычислять значения $w(i,j)$ "с нуля", для чего потребуется $\Theta(j-i)$ операций сложения, будем сохранять эти значения в таблице $w[1..n+1, 0..n]$. В базовом случае вычисляются величины $w[i,i-1] = q_{i-1}$ для $1 \leq i \leq n+1$. Для $j \geq i$ вычисляются величины

$$w[i,j] = w[i,j-1] + p_j + q_j \; . \qquad (15.15)$$

Таким образом, каждое из $\Theta(n^2)$ значений матрицы $w[i,j]$ можно вычислить за время $\Theta(1)$.

Ниже приведен псевдокод, который принимает в качестве входных данных вероятности $p_1, \ldots, p_n$ и $q_0, \ldots, q_n$ и размер n и возвращает таблицы e и $root$.

```
OPTIMAL-BST(p, q, n)
  e[1..n+1, 0..n], w[1..n+1, 0..n]
          и root[1..n, 1..n] — новые таблицы
  for i = 1 to n+1
      e[i, i-1] = q_{i-1}
      w[i, i-1] = q_{i-1}
  for l = 1 to n
      for i = 1 to n-l+1
          j = i+l-1
          e[i, j] = ∞
          w[i, j] = w[i, j-1] + p_j + q_j
          for r = i to j
              t = e[i, r-1] + e[r+1, j] + w[i, j]
              if t < e[i, j]
                  e[i, j] = t
                  root[i, j] = r
  return e и root
```

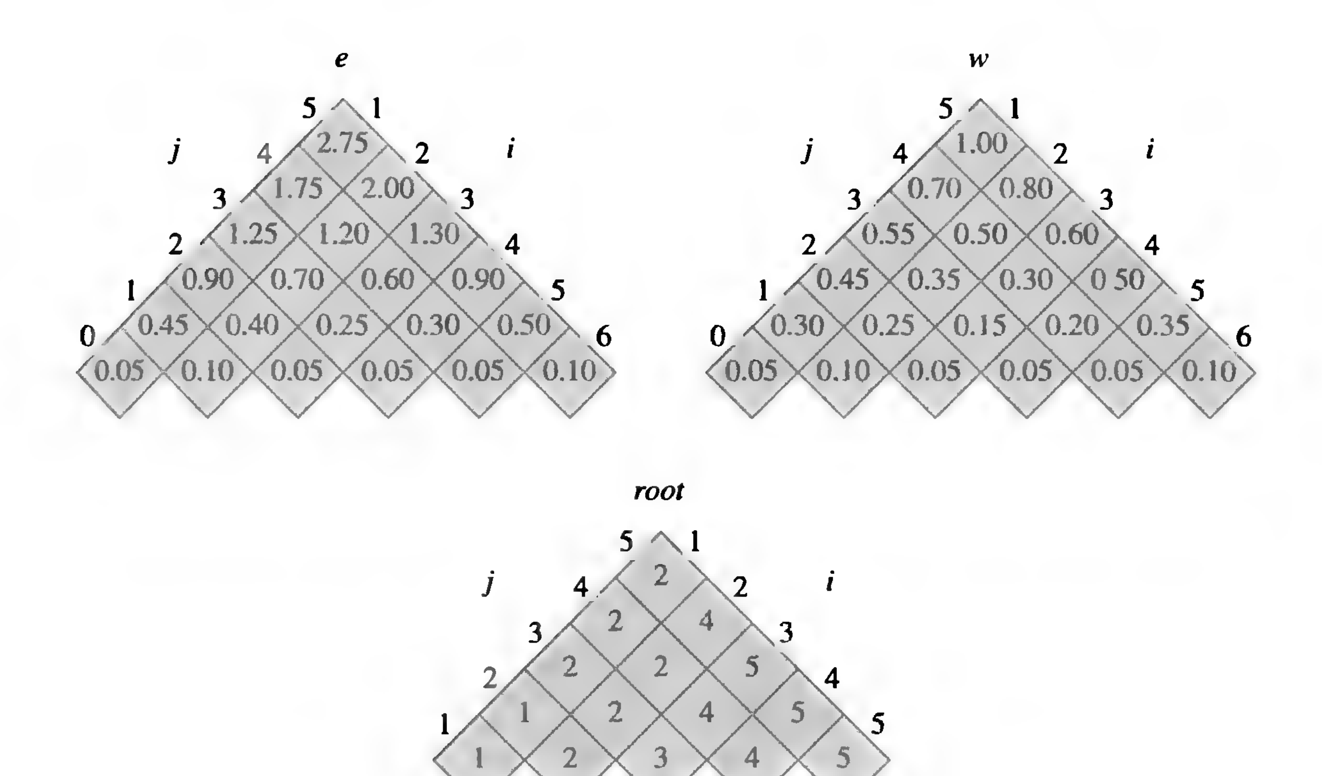


Рис. 15.10. Таблицы $e[i,j]$, $w[i,j]$ и $root[i,j]$, вычисленные процедурой OPTIMAL-BST для распределения ключей, показанного на рис. 15.9. Таблицы повернуты таким образом, чтобы диагонали располагались по горизонтали.

Благодаря приведенному выше описанию и схожести с процедурой MATRIX-CHAIN-ORDER из раздела 15.2, работа представленной выше процедуры должна быть понятной. Цикл **for** в строках 2–4 инициализирует значения $e[i, i-1]$ и $w[i, i-1]$. Затем в цикле **for** в строках 5–14 с помощью рекуррентных соотношений (15.14) и (15.15) вычисляются элементы матриц $e[i,j]$ и $w[i,j]$ для всех индексов $1 \le i \le j \le n$. В первой итерации, когда $l = 1$, в этом цикле вычисляются элементы $e[i,i]$ и $w[i,i]$ для $i = 1, 2, \ldots, n$. Во второй итерации, когда $l = 2$, вычисляются элементы $e[i, i+1]$ и $w[i, i+1]$ для $i = 1, 2, \ldots, n-1$ и т.д. Во внутреннем цикле **for** (строки 10–14) каждый индекс r апробируется на роль индекса корневого элемента k_r оптимального бинарного дерева поиска с ключами $k_i, \ldots, k_j$. В этом цикле элементу $root[i,j]$ присваивается то значение индекса r, которое подходит лучше всего.

На рис. 15.10 показаны таблицы $e[i,j]$, $w[i,j]$ и $root[i,j]$, вычисленные с помощью процедуры OPTIMAL-BST для распределения ключей, показанного на рис. 15.9. Как и в примере с перемножением цепочки матриц на рис. 15.5, таблицы повернуты так, чтобы диагонали располагались горизонтально. В процедуре OPTIMAL-BST строки вычисляются снизу вверх, а в каждой строке заполнение элементов выполняется слева направо.

Время работы процедуры OPTIMAL-BST, как и время работы процедуры MATRIX-CHAIN-ORDER, равно $\Theta(n^3)$. Легко увидеть, что время работы составляет $O(n^3)$, поскольку циклы **for** этой процедуры трижды вложены друг в друга, и индекс каждого цикла принимает не более n значений. Далее, индексы циклов в процедуре OPTIMAL-BST изменяются не в тех же пределах, что и индексы

циклов в процедуре MATRIX-CHAIN-ORDER, но во всех направлениях они принимают по крайней мере одно значение. Таким образом, процедура OPTIMAL-BST, как и процедура MATRIX-CHAIN-ORDER, выполняется в течение времени $\Omega(n^3)$.

Упражнения

15.5.1

Напишите псевдокод процедуры CONSTRUCT-OPTIMAL-BST($root$), которая по заданной таблице $root$ выводит структуру оптимального бинарного дерева поиска. Для примера, приведенного на рис. 15.10, процедура должна выводить структуру, соответствующую оптимальному бинарному дереву поиска, показанному на рис. 15.9, (б).

k_2 является корнем
k_1 является левым дочерним узлом k_2
d_0 является левым дочерним узлом k_1
d_1 является правым дочерним узлом k_1
k_5 является правым дочерним узлом k_2
k_4 является левым дочерним узлом k_5
k_3 является левым дочерним узлом k_4
d_2 является левым дочерним узлом k_3
d_3 является правым дочерним узлом k_3
d_4 является правым дочерним узлом k_4
d_5 является правым дочерним узлом k_5

15.5.2

Определите стоимость и структуру оптимального бинарного дерева поиска для множества, состоящего из $n = 7$ ключей, которым соответствуют следующие вероятности.

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

15.5.3

Предположим, что вместо того, чтобы поддерживать таблицу $w[i, j]$, значение $w(i, j)$ вычисляется в строке 9 процедуры OPTIMAL-BST непосредственно из уравнения (15.12) и используется в строке 11. Как это изменение повлияет на асимптотическое поведение времени выполнения этой процедуры?

15.5.4 ★

Кнут [211] показал, что всегда существуют корни оптимальных поддеревьев, такие, что $root[i, j-1] \leq root[i, j] \leq root[i+1, j]$ для всех $1 \leq i < j \leq n$. Используйте этот факт для модификации процедуры OPTIMAL-BST, при которой время ее работы станет равным $\Theta(n^2)$.

Задачи

15.1. Наидлиннейший простой путь в ориентированном ациклическом графе

Предположим, что имеется ориентированный ациклический граф $G = (V, E)$ с действительными весами ребер и двумя различными вершинами s и t. Опишите подход динамического программирования для поиска наидлиннейшего взвешенного простого пути от s к t. Как выглядит граф подзадач? Какова эффективность разработанного вами алгоритма?

15.2. Наидлиннейшая палиндромная подпоследовательность

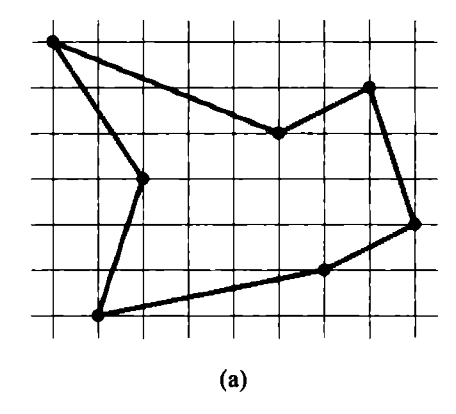
Палиндром представляет собой непустую строку на некотором алфавите, которая одинаково читается в прямом и обратном направлениях. Примерами палиндромов могут служить все строки длиной 1, `потоп`, `топот`, `заказ` и др.

Разработайте эффективный алгоритм поиска наидлиннейшего палиндрома, являющегося подпоследовательностью данной входной строки. Например, для строки `математика` ваш алгоритм должен выводить `атата`. Каково время работы вашего алгоритма?

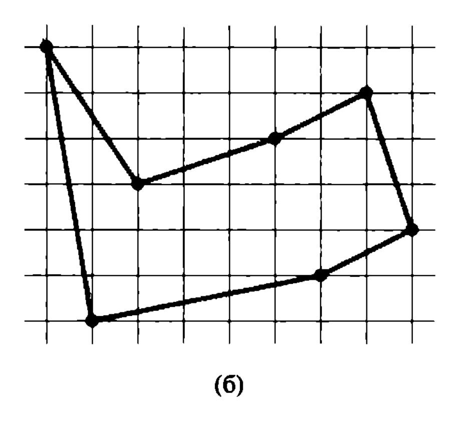
15.3. Битоническая евклидова задача о коммивояжере

В ***битонической евклидовой задаче о коммивояжере*** на плоскости задано множество из n точек, и необходимо найти кратчайший замкнутый путь, соединяющий все эти точки. На рис. 15.11, (а) показано решение задачи, в которой имеется семь точек. В общем случае задача является NP-полной, поэтому считается, что для ее решения требуется время, превышающее полиномиальное (см. главу 34).

Й.Л. Бентли (J.L. Bentley) предположил, что задача упрощается благодаря ограничению интересующих нас маршрутов ***битоническими*** (bitonic tour), т.е. такими, которые начинаются в крайней слева точке, проходят слева направо, а затем — справа налево, возвращаясь прямо к исходной точке. На рис. 15.11, (б) показан кратчайший битонический маршрут, проходящий через те же семь точек. В этом



(а)



(б)

Рис. 15.11. Семь точек на плоскости, наложенные на единичную сетку. **(а)** Кратчайший замкнутый путь длиной около 24.89. Этот путь не является битоническим. **(б)** Кратчайший битонический замкнутый путь для того же множества точек. Его длина — около 25.58.

случае возможна разработка алгоритма, время работы которого является полиномиальным.

Разработайте алгоритм для определения оптимального битонического маршрута, время работы которого будет равным $O(n^2)$. Предполагается, что не существует двух точек, координаты x которых совпадают. (*Указание:* перебор вариантов следует организовать слева направо, поддерживая оптимальные возможности для двух частей, из которых состоит маршрут.)

15.4. Вывод на печать с форматированием

Рассмотрим задачу о форматировании параграфа текста, предназначенного для вывода на печать. Входной текст представляет собой последовательность, состоящую из n слов, длины которых равны $l_1, l_2, \ldots, l_n$ (длина слов измеряется в символах). При выводе на печать параграф должен быть аккуратно отформатирован и размещен в несколько строк, в каждой из которых помещается по M символов. Сформулируем критерий "аккуратного" форматирования следующим образом. Если данная строка содержит слова от i-го до j-го, где $i \leq j$, и мы оставляем между словами ровно по одному пробелу, количество оставшихся пробелов в конце строки, равное $M - j + i - \sum_{k=i}^{j} l_k$, должно быть неотрицательным, чтобы все слова поместились в строке. Мы хотим минимизировать сумму возведенных в куб остатков по всем строкам, кроме последней. Сформулируйте алгоритм, основанный на принципах динамического программирования, который аккуратно выводит на принтер параграф, состоящий из n слов. Проанализируйте время работы этого алгоритма и требуемое для его работы количество памяти.

15.5. Расстояние редактирования

Для того чтобы преобразовать исходную строку текста $x[1..m]$ в конечную строку $y[1..n]$, можно воспользоваться различными операциями преобразования. Наша цель заключается в том, чтобы для данных строк x и y определить последовательность преобразований, превращающих x в y. Для хранения промежуточных результатов используется массив z (предполагается, что он достаточно велик для хранения необходимых промежуточных результатов). Изначально массив z пуст, а в конце работы $z[j] = y[j]$ для всех $j = 1, 2, \ldots, n$. Поддерживаются текущий индекс i массива x и индекс j массива z, и в ходе выполнения операций преобразования разрешено изменять элементы массива z и эти индексы. Изначально $i = j = 1$. В процессе преобразования необходимо проверить каждый символ массива x. Это означает, что в конце выполнения последовательности операций значение i должно быть равно $m + 1$.

Всего имеется шесть перечисленных ниже операций преобразования.

Копирование символа из массива x в массив z путем присвоения $z[j] = x[i]$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Замена символа из массива x другим символом c путем присвоения $z[j] = c$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Удаление символа из массива x путем увеличения на единицу индекса i и сохранения индекса j прежним. В этой операции проверяется элемент $x[i]$.

Вставка символа c в массив z путем присвоения $z[j] = c$ и увеличения на единицу индекса j при сохранении индекса i прежним. В этой операции не проверяется ни один элемент массива x.

Перестановка двух следующих символов путем их копирования из массива x в массив z в обратном порядке. Это делается путем присваиваний $z[j] = x[i+1]$ и $z[j+1] = x[i]$ с последующим изменением значений индексов $i = i+2$ и $j = j+2$. В этой операции проверяются элементы $x[i]$ и $x[i+1]$.

Удаление остатка массива x путем присвоения $i = m+1$. В этой операции проверяются все элементы массива x, которые не были проверены до сих пор. Если эта операция выполняется, то она должна быть последней.

В качестве примера приведем один из способов преобразования исходной строки `algorithm` в конечную строку `altruistic`. В ходе этого преобразования выполняется перечисленная ниже последовательность операций, в которых подчеркнутые символы представляют собой элементы $x[i]$ и $z[j]$ после выполнения очередной операции.

Операция	x	z
Начальные строки	algorithm	_
Копирование	algorithm	a_
Копирование	algorithm	al_
Замена на t	algorithm	alt_
Удаление	algorithm	alt_
Копирование	algorithm	altr_
Вставка u	algorithm	altru_
Вставка i	algorithm	altrui_
Вставка s	algorithm	altruis_
Перестановка	algorithm	altruisti_
Вставка c	algorithm	altruistic_
Удаление остатка	algorithm_	altruistic_

Заметим, что существует ряд других последовательностей операций, позволяющих преобразовать строку `algorithm` в строку `altruistic`.

С каждой операцией преобразования связана своя стоимость, которая зависит от конкретного приложения. Однако мы предположим, что стоимость каждой операции — известная константа. Кроме того, предполагается, что стоимости отдельно взятых операций копирования и замены меньше суммарной стоимости операций удаления и вставки, иначе применять эти операции было бы нерационально. Стоимость некоторой последовательности операций преобразования представляет собой сумму стоимостей отдельных операций последовательности. Стоимость приведенной выше последовательности преобразований строки `algorithm` в строку `altruistic` равна сумме трех стоимостей копирования, стоимости замены, стоимости удаления, четырем стоимостям вставки, стоимости перестановки и удаления остатка.

а. Пусть имеются две последовательности, $x[1..m]$ и $y[1..n]$, а также множество, элементами которого являются стоимости операций преобразования. ***Расстоянием редактирования*** (edit distance) между x и y называется минимальная стоимость последовательности операций преобразования x в y. Опишите основанный на принципах динамического программирования алгоритм, в котором определяется расстояние редактирования от $x[1..m]$ до $y[1..n]$ и выводится оптимальная последовательность операций редактирования. Проанализируйте время работы этого алгоритма и требуемую для него память.

Задача о расстоянии редактирования — это обобщение задачи об анализе двух ДНК (см., например, книгу Сетубала (Setubal) и Мейданиса (Meidanis) [308, раздел 3.2]). Имеется несколько методов, позволяющих оценить подобие двух ДНК путем их выравнивания. Один такой способ выравнивания двух последовательностей x и y заключается в том, чтобы вставлять пробелы в произвольных местах этих двух последовательностей (включая позиции в концах). Получившиеся в результате последовательности x' и y' должны иметь одинаковую длину, однако они не могут содержать пробелы в одинаковых позициях (т.е. не существует такого индекса j, чтобы и $x'[j]$, и $y'[j]$ были пробелами). Затем каждой позиции присваивается определенное количество баллов в соответствии со сформулированными ниже правилами:

- $+1$, если $x'[j] = y'[j]$ и ни один из этих элементов не является пробелом;
- -1, если $x'[j] \neq y'[j]$ и ни один из этих элементов не является пробелом;
- -2, если либо $x'[j]$, либо $y'[j]$ является пробелом.

Стоимость выравнивания определяется как сумма баллов, полученных при сравнении отдельных позиций. Например, если заданы последовательности x = GATCGGCAT и y = CAATGTGAATC, то одно из возможных выравниваний имеет следующий вид.

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Символ + под позицией указывает на то, что ей присваивается балл $+1$, символ - означает балл -1, а символ * — балл -2. Таким образом, полная стоимость равна $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

б. Объясните, как свести задачу о поиске оптимального выравнивания к задаче о поиске расстояния редактирования с помощью подмножества операций преобразования, состоящего из копирования, замены, удаления элемента, вставки, перестановки и удаления остатка.

15.6. Вечеринка на фирме

Профессор Тамада является консультантом президента корпорации по вопросам проведения вечеринок компании. Взаимоотношения между сотрудниками компании описываются иерархической структурой. Это означает, что отношение

"начальник–подчиненный" образует дерево, во главе (в корне) которого находится президент. В отделе кадров каждому служащему присвоили рейтинг дружелюбия, представленный действительным числом. Чтобы на вечеринке все чувствовали себя раскованно, президент выдвинул требование, согласно которому ее не должны одновременно посещать сотрудник и его непосредственный начальник.

Профессору Тамаде предоставили дерево, описывающее структуру корпорации в представлении с левым дочерним и правым сестринским узлами, описанном в разделе 10.4. Каждый узел дерева, кроме указателей, содержит имя сотрудника и его рейтинг дружелюбия. Опишите алгоритм, предназначенный для составления списка гостей, который бы давал максимальное значение суммы рейтингов дружелюбия гостей. Проанализируйте время работы этого алгоритма.

15.7. Алгоритм Витерби

Динамическое программирование можно использовать в ориентированном графе $G = (V, E)$ для распознавания речи. Каждое ребро $(u, v) \in E$ графа помечено звуком $\sigma(u, v)$, который является членом конечного множества звуков Σ. Граф с метками представляет собой формальную модель речи человека, который разговаривает на языке, состоящем из ограниченного множества звуков. Все пути в графе, которые начинаются в выделенной вершине $v_0 \in V$, соответствуют возможной последовательности звуков, допустимых в данной модели. Метка направленного пути по определению является конкатенацией меток, соответствующих ребрам, из которых состоит путь.

а. Разработайте эффективный алгоритм, который для заданного графа G с помеченными ребрами и выделенной вершиной v_0 и для последовательности звуков $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ из множества Σ возвращал бы путь в графе G, который начинается в вершине v_0 и имеет метку s (если таковой существует). В противном случае алгоритм должен возвращать строку NO-SUCH-PATH (нет такого пути). Проанализируйте время работы алгоритма. (*Указание:* могут оказаться полезными концепции, изложенные в главе 22.)

Теперь предположим, что каждому ребру $(u, v) \in E$ также сопоставляется неотрицательная вероятность $p(u, v)$ перехода по этому ребру из вершины u. При этом издается соответствующий звук. Сумма вероятностей по всем ребрам, исходящим из произвольной вершины, равна 1. По определению вероятность пути равна произведению вероятностей составляющих его ребер. Вероятность пути, берущего начало в вершине v_0, можно рассматривать как вероятность "случайного блуждания", которое начинается в этой вершине и продолжается по случайному пути. Выбор каждого ребра на этом пути в вершине u производится в соответствии с вероятностями ребер, исходящих из этой вершины.

б. Обобщите сформулированный в п. (а) ответ так, чтобы возвращаемый путь (если он существует) был *наиболее вероятным* из всех путей, начинающихся в вершине v_0 и имеющих метку s. Проанализируйте время работы этого алгоритма.

15.8. Масштабирование изображения

У нас имеется цветное изображение, состоящее из $m \times n$-массива $A[1\,..\,m, 1\,..\,n]$ пикселей, где каждый пиксель определяется тройкой интенсивностей красного, зеленого и синего цветов (RGB). Предположим, что мы хотим немного сжать это изображение, а именно — удалить по одному пикселю из каждой из m строк, чтобы изображение стало на один пиксель у́же. Во избежание визуальных искажений мы требуем, чтобы пиксели, удаляемые в смежных строках, находились в одном и том же или в смежных столбцах; удаленные пиксели формируют "шов" от верхней до нижней строки, в котором последовательные пиксели шва смежны по вертикали или диагонали.

а. Покажите, что количество таких возможных швов растет как минимум экспоненциально с ростом m, в предположении $n > 1$.

б. Предположим теперь, что для каждого пикселя $A[i,j]$ мы вычисляем действительную меру искажений $d[i,j]$, указывающую, насколько важен тот или иной пиксель $A[i,j]$ в плане удаления. Интуитивно понятно, что чем меньше эта мера для данного пикселя, тем больше он похож на соседние с ним. Затем предположим, что мы определяем меру искажений шва как сумму мер искажений составляющих его пикселей.

Разработайте алгоритм поиска шва с наименьшей мерой искажений. Насколько эффективен ваш алгоритм?

15.9. Разбиение строки

Некоторые языки, ориентированные на работу со строками, позволяют программисту разбивать строки на две части. Поскольку эта операция копирует строки, ее стоимость составляет n единиц времени для разбиения строки из n символов на две части. Предположим, что программист хочет разбить строку на несколько частей. Порядок разбиения может влиять на требуемое для него время. Предположим, например, что программист желает разбить 20-символьную строку после символов с номерами 2, 8 и 10 (используется нумерация в возрастающем порядке слева направо, начиная с 1). Если разбиения выполняются слева направо, то первое сто́ит 20 единиц времени, второе — 18 (строка от символа 3 до символа 20 разбивается после символа 8), а третье — 12 единиц, т.е. общая стоимость равна 50. Если же разбиения выполняются справа налево, то их стоимости соответственно равны 20, 10 и 8 единиц времени, так что общая стоимость равна 38 единиц времени. При еще одном порядке можно сначала разбить строку после символа 8 (стоимость 20), а затем разбить полученные части после символа 2 (стоимость 8) и символа 10 (стоимость 12) с общей стоимостью разбиения, равной 40.

Разработайте алгоритм, который для данных номеров символов, после которых выполняется разбиение, определяет последовательность разбиений с минимальной общей стоимостью. Более формализованно — для заданной строки S длиной n и массива $L[1\,..\,m]$ с точками разбиений вычислить наименьшую стои-

мость последовательности разбиений, а также саму последовательность, при которой достигается эта стоимость.

15.10. Планирование стратегии капиталовложений

Ваши знания алгоритмов помогли вам получить потрясающую работу в компании "КрупноТвердь" с единовременным пособием в 10 000 долларов. Вы решили вложить эти деньги так, чтобы получить за 10 лет как можно больший доход, и обратились для этого в компанию "МЖ инвест", которая предлагает следующие правила. Компания предлагает вам n различных вариантов капиталовложений, пронумерованных от 1 до n. В год j капиталовложение i обеспечивает доход r_{ij}. Другими словами, если вы вложите d долларов в i в год j, то в конце года j будете иметь dr_{ij} долларов. Компания гарантирует, что в течение 10 лет ее президент не сбежит, прихватив кассу, и она будет гарантированно выплачивать ваш доход. Принимать решения о том, куда вложить деньги, вы можете один раз в год. В конце каждого года вы можете либо оставить все деньги там же, где они были в предыдущем году, либо перераспределить их между другими вложениями. Если вы не трогаете свои деньги по окончании года, то платите сбор f_1 долларов; если переносите их — то платите сбор $f_2 > f_1$.

а. Как указано, вы можете вкладывать деньги в несколько предприятий ежегодно. Докажите, что существует оптимальная стратегия, и заключается она в том, чтобы каждый год вкладывать все деньги в одно предприятие. (Напомним, что оптимальная стратегия должна максимизировать количество денег по прошествии 10 лет и не рассматривает никакие иные цели, такие, как, например, минимизация рисков.)

б. Докажите, что задача планирования оптимальной стратегии капиталовложений демонстрирует оптимальную подструктуру.

в. Разработайте алгоритм выработки оптимальной стратегии. Каково время работы вашего алгоритма?

г. Предположим, что "МЖ инвест" добавляет ограничение, заключающееся в том, что в одно предприятие не может быть вложено более 15 000 долларов. Покажите, что при этом ограничении задача максимизации прибыли по истечении 10 лет не обладает оптимальной подструктурой.

15.11. Разработка производственного плана

Компания "Эх, прокачу" изготовляет сани. Естественно, что спрос на эту продукцию колеблется от месяца к месяцу, так что компании требуется разработать стратегию производства с учетом колеблющегося, но предсказуемого спроса. Компании нужен план на следующие n месяцев. Для каждого месяца i известен спрос d_i, т.е. количество саней, которые будут проданы. Пусть $D = \sum_{i=1}^{n} d_i$ — общий спрос саней на все n месяцев. Постоянный штат работников позволяет производить до m саней в месяц. Если требуется произвести большее количество саней, можно нанять временных работников, оплата которых составляет c

долларов за сани. Кроме того, если в конце месяца остаются нераспроданные сани, приходится платить за их хранение. Стоимость хранения j саней представляет собой функцию $h(j)$ для $j = 1, 2, \ldots, D$, где $h(j) \geq 0$ для $1 \leq j \leq D$ и $h(j) \leq h(j+1)$ для $1 \leq j \leq D-1$.

Разработайте алгоритм для расчета производственного плана компании, минимизирующего затраты при выполнении всех ограничений. Время работы алгоритма должно полиномиально зависеть от n и D.

Заключительные замечания

Р. Беллман (R. Bellman) приступил к систематическому изучению динамического программирования в 1955 году. Как в названии "динамическое программирование", так и в термине "линейное программирование" слово "программирование" относится к методу табличного решения. Методы оптимизации, включающие в себя элементы динамического программирования, были известны и раньше, однако именно Беллман дал строгое математическое обоснование этой области [36].

Галил (Galil) и Парк (Park) [124] классифицировали алгоритмы динамического программирования в соответствии с размером таблиц и количеством записей, от которых зависит каждый элемент таблицы. Они называют алгоритм динамического программирования tD/eD, если размер его таблицы равен $O(n^t)$, а каждый ее элемент зависит от $O(n^e)$ других элементов. Например, алгоритм перемножения цепочки матриц из раздела 15.2 является алгоритмом $2D/1D$, а алгоритм поиска наидлиннейшей подпоследовательности из раздела 15.4 — алгоритмом $2D/0D$.

Ху (Hu) и Шинг (Shing) [181,182] разработали алгоритм, позволяющий решить задачу о перемножении матриц в течение времени $O(n \lg n)$.

По-видимому, алгоритм решения задачи о наидлиннейшей общей подпоследовательности, время работы которого равно $O(mn)$, — плод "народного творчества". Кнут (Knuth) [69] поставил вопрос о том, существует ли алгоритм решения этой задачи, время работы которого возрастало бы медленнее, чем квадрат размера. Масек (Masek) и Патерсон (Paterson) [242] ответили на этот вопрос утвердительно, разработав алгоритм, время работы которого равно $O(mn/\lg n)$, где $n \leq m$, а последовательности составлены из элементов множества ограниченного размера. Шимански (Szimanski) [324] показал, что в частном случае, когда ни один элемент не появляется во входной последовательности более одного раза, эту задачу можно решить за время $O((n+m)\lg(n+m))$. Многие из этих результатов были обобщены для задачи о вычислении расстояния редактирования (задача 15.5).

Ранняя работа Гильберта (Gilbert) и Мура (Moore) [132], посвященная бинарному кодированию переменной длины, нашла применение при конструировании оптимального бинарного дерева поиска для случая, когда все вероятности p_i равны 0. В этой работе приведен алгоритм, время работы которого равно $O(n^3)$. Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] представили алгоритм,

описанный в разделе 15.5. Упр. 15.5.4 предложено Кнутом [211]. Ху и Такер (Tucker) [183] разработали алгоритм для случая, когда все вероятности p_i равны 0; время работы этого алгоритма равно $O(n^2)$, а количество необходимой для него памяти — $O(n)$. Впоследствии Кнуту [210] удалось уменьшить это время до величины $O(n \lg n)$.

Задача 15.8 принадлежит Авидану (Avidan) и Шамиру (Shamir) [26], которые разместили в вебе чудесную видеоиллюстрацию этого метода сжатия изображений.

Глава 16. Жадные алгоритмы

Алгоритмы, предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов. Определение наилучшего выбора с помощью принципов динамического программирования во многих задачах оптимизации напоминает стрельбу из пушки по воробьям; другими словами, для этих задач лучше подходят более простые и эффективные алгоритмы. В ***жадном алгоритме*** (greedy algorithm) всегда делается выбор, который кажется самым лучшим в данный момент, т.е. проводится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи. В этой главе рассматриваются задачи оптимизации, для которых жадные алгоритмы приводят к оптимальным решениям. Прежде чем приступить к изучению главы, следует ознакомиться с динамическим программированием, изложенным в главе 15, в частности с разделом 15.3.

Жадные алгоритмы не всегда приводят к оптимальным решениям, но во многих задачах дают нужный результат. В разделе 16.1 рассматривается простая, но нетривиальная задача о выборе процессов, эффективное решение которой можно найти с помощью жадного алгоритма. Чтобы прийти к жадному алгоритму, сначала будет рассмотрено решение, основанное на подходе динамического программирования, после чего будет показано, что оптимальное решение можно получить, исходя из принципов жадных алгоритмов. В разделе 16.2 представлен обзор основных элементов подхода, в соответствии с которым разрабатываются жадные алгоритмы. Это позволяет упростить доказательство корректности жадных алгоритмов. В разделе 16.3 приводится важное приложение методов жадного программирования: разработка кодов Хаффмана (Huffman) для сжатия данных. В разделе 16.4 исследуются теоретические положения, на которых основаны комбинаторные структуры, известные под названием "матроиды", для которых жадный алгоритм всегда дает оптимальное решение. Наконец в разделе 16.5 матроиды применяются для решения задачи о составлении расписания заданий равной длительности с крайним сроком выполнения и штрафами.

Жадный метод обладает достаточной мощью и хорошо подходит для довольно широкого класса задач. В последующих главах представлены многие алгоритмы, которые можно рассматривать как применение жадного метода, включая алгоритмы поиска минимальных остовных деревьев (minimum-spanning-tree) (глава 23), алгоритм Дейкстры (Dijkstra) для определения кратчайших путей из одного ис-

точника (глава 24) и эвристический жадный подход Чватала (Chvatal) к задаче о покрытии множества (set-covering) (глава 35). Алгоритмы поиска минимальных остовных деревьев являются классическим примером применения жадного метода. Несмотря на то что эту главу и главу 23 можно читать независимо одна от другой, может оказаться полезным прочитать их вместе.

16.1. Задача о выборе процессов

В качестве первого примера рассмотрим задачу о составлении расписания для нескольких конкурирующих процессов, каждый из которых безраздельно использует общий ресурс. Цель этой задачи — выбор набора взаимно совместимых процессов, образующих множество максимального размера. Предположим, имеется множество $S = \{a_1, a_2, \ldots, a_n\}$, состоящее из n ***процессов*** (activities). Процессам требуется некоторый ресурс, который одновременно может использоваться лишь одним процессом. Каждый процесс a_i характеризуется ***начальным моментом*** s_i и ***конечным моментом*** f_i, где $0 \leq s_i < f_i < \infty$. Будучи выбранным, процесс a_i длится в течение полуоткрытого интервала времени $[s_i, f_i)$. Процессы a_i и a_j ***совместимы*** (compatible), если интервалы $[s_i, f_i)$ и $[s_j, f_j)$ не перекрываются (т.е. если $s_i \geq f_j$ или $s_j \geq f_i$). ***Задача о выборе процессов*** (activity-selection problem) заключается в том, чтобы выбрать максимальное по размеру подмножество взаимно совместимых процессов. Например, рассмотрим описанное ниже множество S процессов, отсортированных в порядке возрастания моментов окончания:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n \ . \tag{16.1}$$

(Вскоре станет понятно, почему удобнее рассматривать процессы, расположенные именно в таком порядке.) Например, рассмотрим следующее множество процессов S.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

В данном примере из взаимно совместимых процессов можно составить подмножество $\{a_3, a_9, a_{11}\}$. Однако оно не является максимальным, поскольку существует большее подмножество $\{a_1, a_4, a_8, a_{11}\}$. Фактически подмножество $\{a_1, a_4, a_8, a_{11}\}$ — наибольшее подмножество взаимно совместимых процессов. Еще одно такое подмножество — $\{a_2, a_4, a_9, a_{11}\}$.

Разобьем решение этой задачи на несколько этапов. Начнем с того, что сформулируем решение рассматриваемой задачи, основанное на принципах динамического программирования. Это оптимальное решение исходной задачи получается путем комбинирования оптимальных решений подзадач. Мы рассмотрим несколько вариантов выбора, который делается в процессе определения подзадачи, использующейся в оптимальном решении. Затем станет понятно, что заслуживает внимания лишь один выбор — жадный — и что, когда делается этот выбор, од-

15 Зак. 3726

на из подзадач гарантированно получается пустой. Остается лишь одна непустая подзадача. Исходя из этих наблюдений, мы разработаем рекурсивный жадный алгоритм, предназначенный для решения задачи о составлении расписания процессов. Процесс разработки жадного алгоритма будет завершен его преобразованием из рекурсивного в итерационный. Описанные в этом разделе этапы сложнее, чем те, которые обычно имеют место при разработке жадных алгоритмов, однако они иллюстрируют взаимоотношение между жадными алгоритмами и динамическим программированием.

Оптимальная подструктура задачи о выборе процессов

Можно легко убедиться в том, что задача выбора процессов демонстрирует оптимальную подструктуру. Обозначим через S_{ij} множество процессов, которые запускаются после завершения процесса a_i и завершаются до запуска процесса a_j. Предположим, что необходимо найти максимальное множество взаимно совместимых процессов в S_{ij} и что такое максимальное множество представляет собой A_{ij}, которое включает некоторый процесс a_k. Включая a_k в оптимальное решение, мы оказываемся с двумя подзадачами: поиска взаимно совместимых процессов во множестве S_{ik} (процессов, которые начинаются после завершения процесса a_i и завершаются до начала процесса a_k) и поиска взаимно совместимых процессов во множестве S_{kj} (процессов, которые начинаются после завершения процесса a_k и завершаются до начала процесса a_j). Пусть $A_{ik} = A_{ij} \cap S_{ik}$ и $A_{kj} = A_{ij} \cap S_{kj}$, так что A_{ik} содержит процессы в A_{ij}, которые завершаются до начала a_k, а A_{kj} содержит процессы в A_{ij}, которые начинаются после того, как завершается a_k. Таким образом, мы имеем $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, и множество максимального размера A_{ij} взаимно совместимых процессов в S_{ij} состоит из $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ процессов.

Обычные рассуждения с применением метода "вырезания и вставки" показывают, что оптимальное решение A_{ij} должно включать оптимальные решения двух подзадач для S_{ik} и S_{kj}. Если бы мы могли найти множество A'_{kj} взаимно совместимых процессов в S_{kj}, такое, что $|A'_{kj}| > |A_{kj}|$, то мы могли бы использовать A'_{kj} вместо A_{kj} в решении подзадачи для S_{ij}. Таким образом, мы могли бы построить множество $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ взаимно совместимых задач, что противоречит предположению о том, что A_{ij} является оптимальным решением. Такие же рассуждения применимы и к процессам в S_{ik}.

Этот способ характеристики оптимальной подструктуры дает основания для решения данной задачи методами динамического программирования. Если обозначить размер оптимального решения для множества S_{ij} как $c[i, j]$, то мы получим рекуррентное соотношение

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Конечно, если мы не знаем, что оптимальное решение для множества S_{ij} включает процесс a_k, нам следует проверить все процессы в S_{ij}, чтобы найти, какой

из них должен быть выбран, так что

$$c[i,j] = \begin{cases} 0 , & \text{если } S_{ij} = \emptyset , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} , & \text{если } S_{ij} \neq \emptyset . \end{cases} \quad (16.2)$$

Мы можем разработать рекурсивный алгоритм и применить к нему запоминание либо заполнять таблицу, работая в восходящем направлении. Но можно обратить внимание на еще одну важную характеристику задачи выбора процесса, которая позволяет существенно усовершенствовать решение поставленной задачи.

Осуществление жадного выбора

Что если мы могли бы выбирать процесс для добавления в оптимальное решение, не решая предварительно все подзадачи? Это могло бы спасти нас от необходимости рассматривать все выборы, возникающие в рекуррентном соотношении (16.2). В действительности в задаче выбора процессов нужно рассмотреть только один выбор, а именно — жадный.

Что означает "жадный выбор" применительно к задаче выбора процессов? Интуиция подсказывает нам, что это должен быть выбор, который оставляет ресурсы доступными как можно большему количеству процессов. В выбранном нами множестве процессов один из них должен завершаться первым. Подсказка интуиции заключается в том, что этот процесс должен завершиться как можно раньше, чтобы оставить процессам, выполняющимся после него, как можно больше времени. (Если таких процессов с наиболее ранним временем завершения в S имеется несколько, можно выбрать любой из них.) Другими словами, поскольку процессы отсортированы в возрастающем порядке времени завершения, жадный выбор представляет собой выбор процесса a_1. Выбор процесса, завершающегося первым, — не единственный жадный выбор для поставленной задачи; в упр. 16.1.3 предлагается рассмотреть другие варианты жадного выбора для этой задачи.

Делая жадный выбор, мы остаемся только с одной нерешенной подзадачей: поиска процессов, которые начинаются после завершения a_1. Почему мы не должны рассматривать процессы, завершающиеся до того, как начнется процесс a_1? Мы имеем $s_1 < f_1$, а f_1 — самое раннее время завершения любого процесса. Следовательно, никакие процессы не могут иметь время завершения, меньшее или равное времени s_1. Таким образом, все совместимые с a_1 процессы должны начинаться после завершения процесса a_1.

Кроме того, мы уже установили, что задача выбора процессов демонстрирует оптимальную подструктуру. Пусть $S_k = \{a_i \in S : s_i \geq f_k\}$ является множеством процессов, начинающихся после завершения процесса a_k. Если мы делаем жадный выбор процесса a_1, то нам остается решить единственную подзадачу S_1.[1] Оптимальная подструктура говорит нам о том, что если a_1 представляет собой

---

[1]Мы иногда говорим о множествах S_k как о подзадачах, а не как о множествах процессов. Но из контекста всегда понятно, что именно мы имеем в виду, упоминая S_k.

оптимальное решение, то оптимальное решение исходной задачи состоит из процесса a_1 и всех процессов в оптимальном решении подзадачи S_1.

Остается нерешенным главный вопрос: не подводит ли нас наша интуиция? Всегда ли жадный выбор — при котором мы выбираем процесс, завершающийся первым, — является частью некоторого оптимального решения? Приведенная далее теорема показывает, что это так и есть.

Теорема 16.1

Рассмотрим любую непустую подзадачу S_k, и пусть a_m представляет собой процесс в S_k, завершающийся раньше других. Тогда a_m входит в некоторое подмножество взаимно совместимых процессов S_k максимального размера.

Доказательство. Пусть A_k представляет собой подмножество взаимно совместимых задач S_k, имеющее максимальный размер, и пусть a_j является процессом в A_k с самым ранним временем завершения. Если $a_j = a_m$, доказательство завершено, поскольку мы показали, что a_m находится в некотором подмножестве взаимно совместимых процессов S_k максимального размера. Если $a_j \neq a_m$, пусть множество $A'_k = A_k - \{a_j\} \cup \{a_m\}$ представляет собой множество A_k, в котором a_m заменено на a_j. Процессы в A'_k не перекрываются, что следует из того, что процессы в A_k не перекрываются, a_j является процессом из A_k, завершающимся первым, и $f_m \leq f_j$. Поскольку $|A'_k| = |A_k|$, мы заключаем, что A'_k — подмножество взаимно совместимых процессов S_k максимального размера, и оно включает a_m. ■

Таким образом, мы видим, что хотя задачу выбора процессов можно решать методами динамического программирования, необходимости в этом нет. (Кроме того, нам даже не нужно проверять, имеются ли в задаче выбора процессов перекрывающиеся подзадачи.) Вместо этого можно многократно выполнять выбор процесса, завершающегося первым, оставляя при этом только процессы, совместимые с выбранным, и повторяя эти действия до тех пор, пока не останется ни одного процесса. Более того, поскольку мы всегда выбираем процесс с наиболее ранним временем завершения, времена завершения выбранных процессов должны находиться в строго возрастающем порядке. Мы можем рассматривать каждый процесс однократно, в порядке возрастания времен завершения процессов.

Алгоритму решения задачи выбора процессов нет необходимости работать в восходящем направлении, как это делал бы табличный алгоритм динамического программирования. Вместо этого он может работать в нисходящем направлении, выбирая процесс для помещения в оптимальное решение с последующим решением подзадачи выбора процессов из множества совместимых с уже выбранными. Жадные алгоритмы обычно имеют нисходящий дизайн: они сначала делают выбор, а затем решают подзадачу, в отличие от восходящего метода, который решает подзадачи перед тем, как сделать выбор.

Рекурсивный жадный алгоритм

Теперь, после того как мы увидели, каким образом можно отказаться от решения, основанного на принципах динамического программирования, и вместо этого использовать нисходящий жадный алгоритм, можно написать простую рекурсивную процедуру для решения задачи выбора процессов. Процедура RECURSIVE-ACTIVITY-SELECTOR получает значения начальных и конечных моментов процессов, представленные в виде массивов s и f,[2] а также индекс k, определяющий подзадачу S_k, которую требуется решить, и размер n исходной задачи. Процедура возвращает множество максимального размера, состоящее из взаимно совместимых процессов в S_k. Предполагается, что все n входных процессов уже отсортированы и расположены в порядке монотонного возрастания времени их окончания согласно (16.1). Если это не так, их можно отсортировать в указанном порядке за время $O(n \lg n)$, произвольным образом разрешая могущие возникнуть неоднозначности. Начальный вызов этой процедуры имеет вид RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n и s[m] < f[k]     // Находим первый подходящий
                                 // процесс в S_k
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Работа представленного алгоритма проиллюстрирована на рис. 16.1. В рекурсивном вызове процедуры RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), в цикле **while** в строках 2 и 3, осуществляется поиск первого завершающегося процесса в S_k. В цикле процессы $a_{k+1}, a_{k+2}, \ldots, a_n$ проверяются до тех пор, пока не будет найден первый процесс a_m, совместимый с процессом a_k; для такого процесса справедливо соотношение $s_m \geq f_k$. Если цикл завершается из-за того, что такой процесс найден, то в строке 5 происходит возврат из процедуры объединения $\{a_m\}$ с подмножеством максимального размера для задачи S_m, которое возвращается рекурсивным вызовом RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Еще одна возможная причина завершения процесса — достижение условия $m > n$. В этом случае проверены все процессы в S_k, но среди них не найден такой, который был бы совместим с процессом a_k. В этом случае $S_k = \emptyset$, и процедура возвращает $\emptyset$ в строке 6.

В предположении, что все процессы отсортированы в соответствии со временами их окончания, время работы вызова RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ равно $\Theta(n)$, в чем можно убедиться следующим образом.

---

[2]Поскольку псевдокод получает s и f в виде массивов, они индексируются с применением записи с квадратными скобками, а не нижними индексами.

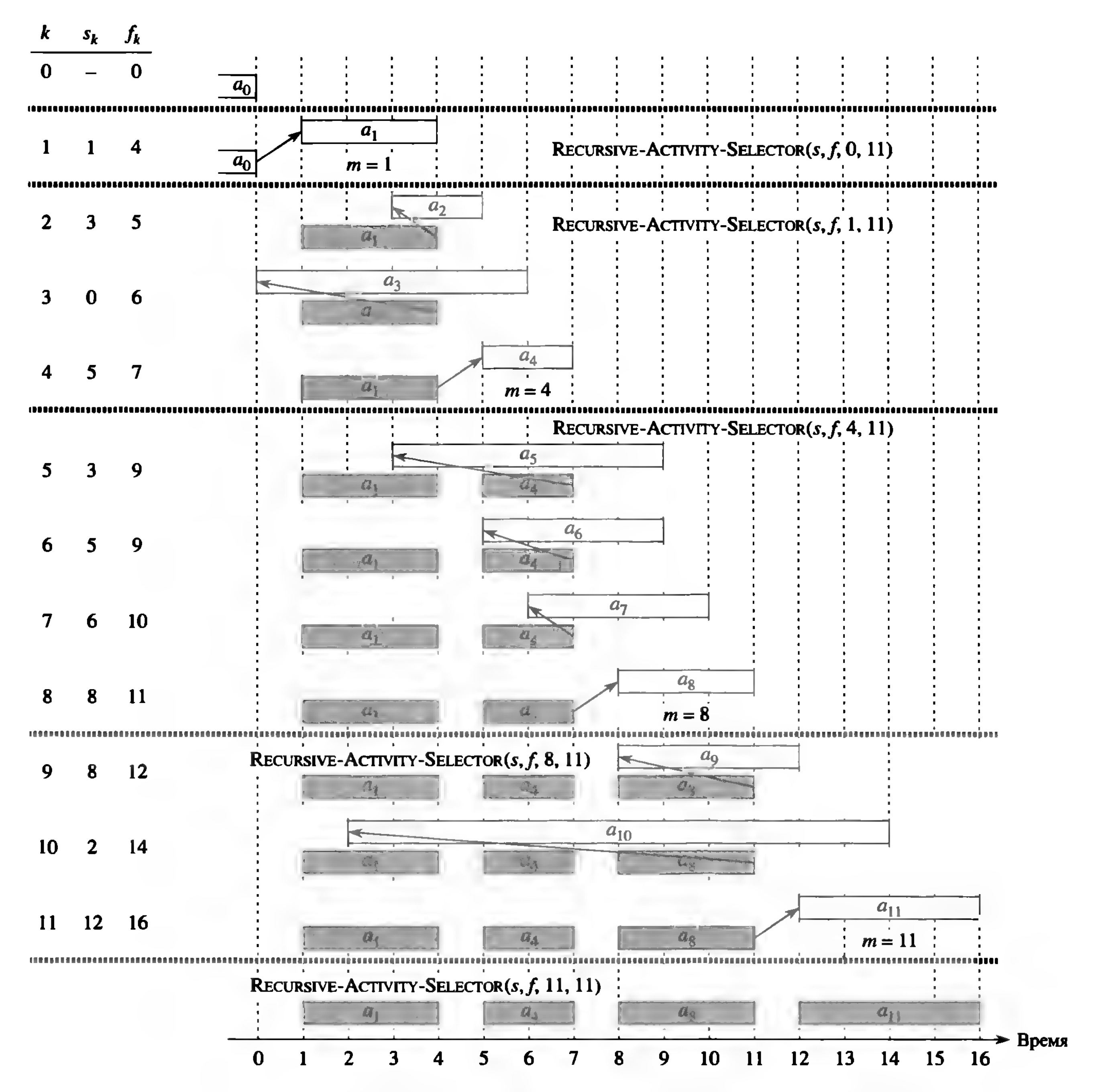


Рис. 16.1. Работа процедуры RECURSIVE-ACTIVITY-SELECTOR с заданными ранее 11 процессами. Процессы, рассматриваемые при каждом рекурсивном вызове, располагаются между горизонтальными линиями. Фиктивный процесс a_0 завершается в момент 0, и начальный вызов RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$ выбирает процесс a_1. В каждом рекурсивном вызове отобранные процессы показаны заштрихованными, а проверяемые процессы показаны белыми прямоугольниками. Если процесс начинается до окончания последнего добавленного процесса (стрелка между этими моментами указывает влево), он отвергается. В противном случае (стрелка указывает вверх или вправо) процесс выбирается. Последний рекурсивный вызов, RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$, возвращает $\emptyset$. Полученное в результате множество процессов имеет вид $\{a_1, a_4, a_8, a_{11}\}$.

Сколько бы ни было рекурсивных вызовов, каждый процесс проверяется в цикле **while**, в строке 2, по одному разу. В частности, процесс a_i проверяется в последнем вызове, когда $k < i$.

Итеративный жадный алгоритм

Представленную ранее рекурсивную процедуру легко преобразовать в итеративную. Процедура RECURSIVE-ACTIVITY-SELECTOR почти подпадает под определение оконечной рекурсии (см. задачу 7.4): она оканчивается рекурсивным вызовом самой себя, после чего выполняется операция объединения. Преобразование процедуры, построенной по принципу оконечной рекурсии, в итерационную — обычно простая задача (некоторые компиляторы разных языков программирования выполняют эту задачу автоматически). Как уже упоминалось, процедура RECURSIVE-ACTIVITY-SELECTOR выполняется для подзадач S_k, т.е. для подзадач, состоящих из процессов, которые оканчиваются позже других.

Процедура GREEDY-ACTIVITY-SELECTOR — итеративная версия процедуры RECURSIVE-ACTIVITY-SELECTOR. В ней также предполагается, что входные процессы расположены в порядке монотонного возрастания времен окончания. Выбранные процессы объединяются в этой процедуре в множество A, которое и возвращается процедурой после ее окончания.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Процедура работает следующим образом. Переменная k индексирует последнее добавление в множество A, соответствующее процессу a_k в рекурсивной версии. Поскольку процессы рассматриваются в порядке монотонного возрастания моментов их окончания, f_k — всегда максимальное время окончания всех процессов, принадлежащих множеству A, т.е.

$$f_k = \max \{f_i : a_i \in A\} \ . \tag{16.3}$$

В строках 2 и 3 выбирается процесс a_1 и инициализируется множество A, содержащее только этот процесс, а переменной k присваивается индекс этого процесса. В цикле **for** в строках 4–7 происходит поиск процесса задачи S_k, оканчивающегося раньше других. В этом цикле по очереди рассматривается каждый процесс a_m, который добавляется в множество A, если он совместим со всеми ранее выбранными процессами; этот процесс оканчивается в S_k раньше других. Чтобы узнать, совместим ли процесс a_m с процессами, которые уже содержатся во множестве A, в соответствии с уравнением (16.3) достаточно проверить (строка 5),

что его начальный момент s_m наступает не раньше момента f_k окончания последнего из добавленных в множество A процессов. Если процесс a_m удовлетворяет сформулированным выше условиям, то в строках 6 и 7 он добавляется в множество A и переменной k присваивается значение m. Множество A, возвращаемое вызовом GREEDY-ACTIVITY-SELECTOR(s, f), в точности совпадает с тем, которое возвращается вызовом RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Процедура GREEDY-ACTIVITY-SELECTOR, как и рекурсивная версия алгоритма, составляет расписание для n-элементного множества в течение времени $\Theta(n)$. Это утверждение справедливо в предположении, что процессы уже отсортированы в порядке возрастания времени их окончания.

Упражнения

16.1.1

Разработайте алгоритм динамического программирования для решения задачи о выборе процессов, основанный на рекуррентном соотношении (16.2). В этом алгоритме должны вычисляться определенные выше размеры $c[i, j]$ и должно выводиться подмножество взаимно совместимых процессов максимального размера. Предполагается, что процессы отсортированы в порядке, заданном уравнением (16.1). Сравните время работы найденного алгоритма со временем работы процедуры GREEDY-ACTIVITY-SELECTOR.

16.1.2

Предположим, что вместо того, чтобы все время выбирать процесс, который оканчивается раньше других, выбирается процесс, который начинается позже других и совместим со всеми ранее выбранными процессами. Опишите этот подход как жадный алгоритм и докажите, что он позволяет получить оптимальное решение.

16.1.3

К максимальному множеству взаимно совместимых процессов приводит не любой жадный подход. Приведите пример, демонстрирующий, что подход, основанный на выборе процессов с минимальным временем работы среди совместимых с уже выбранными процессами, не работает. Выполните то же самое для подходов с выбором совместимого процесса, который перекрывается с наименьшим количеством остающихся процессов или с выбором из доступных процессов с самым ранним временем начала работы.

16.1.4

Предположим, что имеется множество процессов, для которых нужно составить расписание при наличии большого количества ресурсов. Например, это может быть расписание лекций в разных аудиториях, и наша задача — составить расписание лекций, которое бы использовало как можно меньше аудиторий.

(Эта задача известна также как задача о ***раскрашивании графа отрезков*** (interval-graph colouring problem). Можно создать граф отрезков, вершины которого сопоставляются заданным процессам, а ребра соединяют несовместимые процессы. Минимальное количество цветов, необходимых для раскрашивания всех вершин

таким образом, чтобы никакие две соединенные вершины не имели один и тот же цвет, будет равно минимальному количеству ресурсов, необходимых для работы всех заданных процессов.)

16.1.5

Рассмотрим модификацию задачи о выборе процессов, в которой каждый процесс a_i в дополнение ко времени начала и конца работы имеет некоторое значение v_i. Наша цель теперь заключается не в максимизации количества отобранных процессов, а в достижении максимальной суммы их значений. То есть мы хотим выбрать множество A совместимых процессов, такое, что сумма $\sum_{a_k \in A} v_k$ является максимальной. Разработайте алгоритм решения этой задачи с полиномиальным временем работы.

16.2. Элементы жадной стратегии

Жадный алгоритм позволяет получить оптимальное решение задачи путем осуществления ряда выборов. В каждой точке принятия решения в алгоритме делается выбор, который в данный момент выглядит самым лучшим. Эта эвристическая стратегия не всегда дает оптимальное решение, но все же решение может оказаться и оптимальным, в чем мы смогли убедиться на примере задачи о выборе процессов. В настоящем разделе рассматриваются некоторые общие свойства жадных методов.

Процесс разработки жадного алгоритма, рассмотренный в разделе 16.1, несколько сложнее, чем обычно. Были пройдены перечисленные ниже этапы.

1. Определена оптимальная подструктура задачи.
2. Разработано рекурсивное решение. (Для задачи выбора процессов мы сформулировали рекуррентное соотношение (16.2), но пропустили разработку алгоритма на его основе.)
3. Показано, что при жадном выборе у нас остается только одна подзадача.
4. Доказано, что жадный выбор всегда безопасен. (Этапы 3 и 4 могут следовать в любом порядке.)
5. Разработан рекурсивный алгоритм, реализующий жадную стратегию.
6. Рекурсивный алгоритм преобразован в итеративный.

В ходе выполнения этих этапов мы во всех подробностях смогли рассмотреть, как динамическое программирование послужило основой для жадного алгоритма. Например, в задаче о выборе процессов сначала определяются подзадачи S_{ij}, в которых изменяются оба индекса — и i, и j. Затем мы выяснили, что если всегда делается жадный выбор, то подзадачи можно было бы ограничить видом S_k.

Можно предложить и альтернативный подход, в котором оптимальная подструктура приспосабливалась бы специально для жадного выбора, т.е. второй индекс можно было бы опустить и определять подзадачи в виде S_k. Затем можно

было бы доказать, что жадный выбор (процесс a_m, который оканчивается первым в S_k) в сочетании с оптимальным решением для остающегося множества S_m совместимых между собой процессов приводит к оптимальному решению задачи S_k. Обобщая сказанное, опишем процесс разработки жадных алгоритмов в виде последовательности перечисленных далее этапов.

1. Привести задачу оптимизации к виду, когда после сделанного выбора остается решить только одну подзадачу.
2. Доказать, что всегда существует такое оптимальное решение исходной задачи, которое можно получить путем жадного выбора, так что такой выбор всегда допусти́м.
3. Продемонстрировать оптимальную структуру, показав, что после жадного выбора остается подзадача, обладающая тем свойством, что объединение оптимального решения подзадачи со сделанным жадным выбором приводит к оптимальному решению исходной задачи.

Описанный выше более прямой процесс будет использоваться в последующих разделах данной главы. Тем не менее заметим, что в основе каждого жадного алгоритма почти всегда находится более сложное решение в стиле динамического программирования.

Как определить, способен ли жадный алгоритм решить конкретную задачу оптимизации? Общего пути здесь нет, однако можно выделить два ключевых компонента — свойство жадного выбора и оптимальную подструктуру. Если удается продемонстрировать, что задача обладает двумя этими свойствами, то с большой вероятностью для нее можно разработать жадный алгоритм.

Свойство жадного выбора

Первая из названных выше основных составляющих жадного алгоритма — ***свойство жадного выбора***: глобальное оптимальное решение можно получить, делая локально оптимальный (жадный) выбор. Другими словами, рассматривая, какой выбор следует сделать, мы делаем выбор, который кажется самым лучшим в текущей задаче; результаты возникающих подзадач при этом не рассматриваются.

Рассмотрим отличие жадных алгоритмов от динамического программирования. В динамическом программировании на каждом этапе делается выбор, однако обычно этот выбор зависит от решений подзадач. Следовательно, методом динамического программирования задачи обычно решаются в восходящем направлении, т.е. сначала обрабатываются более простые подзадачи, а затем — более сложные. (В качестве альтернативного пути решения можно решать задачу в нисходящем направлении, но с применением запоминания. Конечно же, несмотря на то, что код работает в нисходящем направлении, решения подзадач должны быть найдены до того, как будет сделан соответствующий выбор.) В жадном алгоритме делается выбор, который выглядит в данный момент наилучшим, после чего решается подзадача, возникающая в результате этого выбора. Выбор, сделанный в жадном алгоритме, может зависеть от сделанных ранее выборов, но не от каких

бы то ни было выборов или решений последующих подзадач. Таким образом, в отличие от динамического программирования, в котором подзадачи решаются до выполнения первого выбора, жадный алгоритм делает первый выбор до решения подзадач. Алгоритмы динамического программирования работают в восходящем направлении, жадная же стратегия обычно разворачивается в нисходящем, когда жадный выбор делается один за другим, в результате чего каждый экземпляр текущей задачи сводится к меньшему.

Конечно же, необходимо доказать, что жадный выбор на каждом этапе приводит к глобально оптимальному решению. Обычно, как это было в теореме 16.1, в таком доказательстве исследуется глобально оптимальное решение некоторой подзадачи. Затем демонстрируется, что это решение можно преобразовать так, чтобы вместо некоторого другого в нем использовался жадный выбор, приводящий задачу к аналогичной подзадаче меньшего размера.

Обычно жадный выбор можно делать более эффективно, чем при рассмотрении большого набора выборов. Например, если в задаче о выборе процессов предварительно отсортировать процессы в порядке монотонного возрастания моментов их окончания, то каждый из них достаточно рассмотреть всего один раз. Зачастую оказывается, что благодаря предварительной обработке входных данных или применению подходящей структуры данных (нередко это очередь с приоритетами) можно ускорить процесс жадного выбора, что приведет к повышению эффективности алгоритма.

Оптимальная подструктура

Задача демонстрирует ***оптимальную подструктуру***, если в ее оптимальном решении содержатся оптимальные решения подзадач. Это свойство является основным признаком применимости как динамического программирования, так и жадных алгоритмов. В качестве примера оптимальной подструктуры напомним результаты, полученные в разделе 16.1. В нем было продемонстрировано, что если оптимальное решение подзадачи S_{ij} содержит процесс a_k, то оно также содержит оптимальные решения подзадач S_{ik} и S_{kj}. При наличии этой оптимальной подструктуры было показано, что если известно, какой процесс используется в качестве процесса a_k, то оптимальное решение задачи S_{ij} можно построить путем выбора процесса a_k и его объединения со всеми процессами в оптимальных решениях подзадач S_{ik} и S_{kj}. На основе этого наблюдения удалось получить рекуррентное соотношение (16.2), описывающее оптимальное решение.

Обычно при работе с жадными алгоритмами применяется более простой подход. Как уже упоминалось, мы воспользовались предположением, что подзадача получилась в результате жадного выбора в исходной задаче. Все, что осталось сделать, — это обосновать, что оптимальное решение подзадачи в сочетании с ранее сделанным жадным выбором приводит к оптимальному решению исходной задачи. В этой схеме для доказательства того, что жадный выбор на каждом шаге приводит к оптимальному решению, неявно используется индукция по вспомогательным задачам.

Сравнение жадных алгоритмов и динамического программирования

Поскольку свойство оптимальной подструктуры применяется и в жадных алгоритмах, и в стратегии динамического программирования, может возникнуть соблазн разработать решение в стиле динамического программирования для задачи, в которой достаточно применить жадное решение. Не исключена также возможность ошибочного вывода о применимости жадного решения в той ситуации, когда необходимо решать задачу методом динамического программирования. Чтобы проиллюстрировать тонкие различия между этими двумя методами, рассмотрим две разновидности классической задачи оптимизации.

Дискретная задача о рюкзаке (0-1 knapsack problem) формулируется следующим образом. Вор во время ограбления магазина обнаружил n предметов. Предмет под номером i имеет стоимость v_i и вес w_i, где и v_i, и w_i — целые числа. Нужно унести вещи, суммарная стоимость которых была бы как можно большей, однако грузоподъемность рюкзака ограничена и равна W, где W — целая величина. Какие предметы следует взять с собой? (Предмет не может быть разделен на части или взят более одного раза.)

Ситуация в ***континуальной задаче о рюкзаке*** (fractional knapsack problem) та же, но теперь тот или иной товар вор может брать с собой частично, а не делать каждый раз бинарный выбор — брать или не брать (0–1). В дискретной задаче о рюкзаке в роли предметов могут выступать, например, слитки золота, а для континуальной задачи лучше подходят такие товары, как золотой песок.

В обеих разновидностях задачи о рюкзаке проявляется свойство оптимальной подструктуры. Рассмотрим в дискретной задаче наиболее ценную загрузку, вес которой не превышает W. Если вынуть из рюкзака предмет под номером j, то остальные предметы должны быть наиболее ценными; при этом их вес не должен превышать $W - w_j$ и их можно составить из $n - 1$ исходных предметов, из множества которых исключен предмет под номером j. Для аналогичной континуальной задачи можно привести такие же рассуждения. Если удалить из оптимально загруженного рюкзака часть товара с индексом j, которая весит w, остальное содержимое рюкзака будет наиболее ценным и состоящим из $n - 1$ исходных товаров, вес которых не превышает величину $W - w$ плюс $w_j - w$ кг товара с индексом j.

Несмотря на сходство сформулированных выше задач, континуальная задача о рюкзаке допускает решение на основе жадной стратегии, а дискретная — не допускает. Чтобы решить континуальную задачу, вычислим сначала стоимость единицы веса v_i/w_i каждого товара. Придерживаясь жадной стратегии, вор сначала загружает как можно больше товара с максимальной удельной стоимостью (стоимостью единицы веса). Если запас этого товара исчерпается, а грузоподъемность рюкзака — нет, он загружает как можно больше товара, удельная стоимость которого будет второй по величине. Так продолжается до тех пор, пока вес товара не достигает допустимого максимума. Таким образом, вместе со временем сортировки товаров по их удельной стоимости время работы алгоритма равно $O(n \lg n)$. Доказательство того, что континуальная задача о рюкзаке обладает свойством жадного выбора, предлагается провести в упражнении 16.2.1.

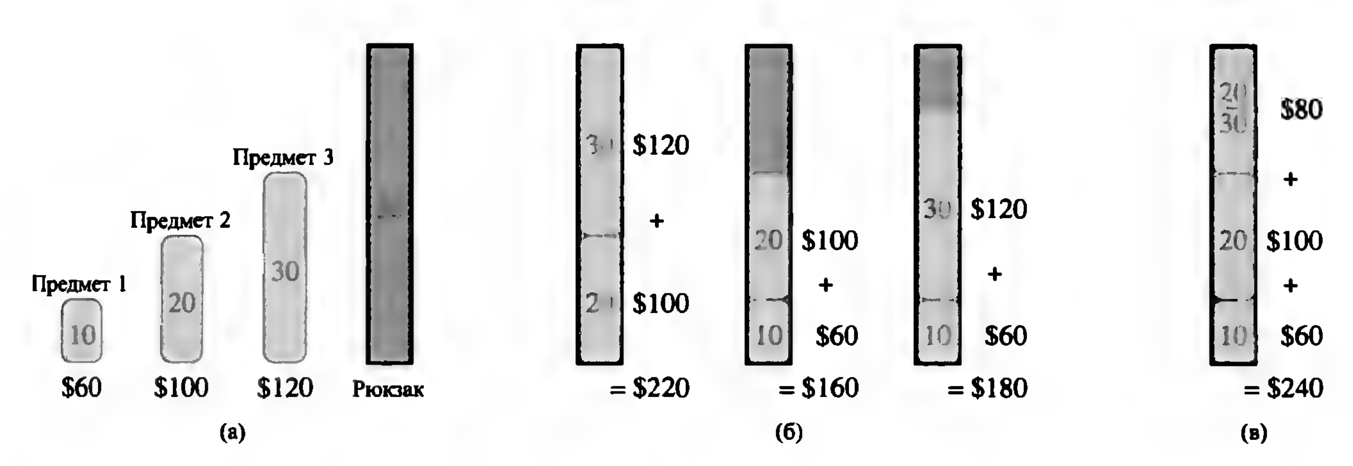


Рис. 16.2. Контрпример, показывающий, что жадная стратегия не работает в дискретной задаче о рюкзаке. **(а)** Вор должен выбрать подмножество трех предметов, вес которых не должен превышать 50 кг. **(б)** Оптимальное подмножество включает предметы 2 и 3. Любое решение с предметом 1 не является оптимальным, несмотря на наивысшую удельную стоимость первого предмета. **(в)** В случае континуальной задачи о рюкзаке выбор товаров с наивысшей удельной стоимостью приводит к оптимальному решению.

Чтобы убедиться, что подобная жадная стратегия не работает в дискретной задаче о рюкзаке, рассмотрим пример, проиллюстрированный на рис. 16.2, (а). Имеется 3 предмета и рюкзак, способный выдержать 50 кг. Первый предмет весит 10 кг и стоит 60 долларов. Второй предмет весит 20 кг и стоит 100 долларов. Третий предмет весит 30 кг и стоит 120 долларов. Таким образом, один килограмм первого предмета стоит 6 долларов, что превышает аналогичную величину для второго (5 долларов/кг) и третьего (4 доллара/кг) предметов. Поэтому жадная стратегия должна состоять в том, чтобы сначала взять первый предмет. Однако, как видно из рис. 16.2, (б), оптимальное решение — взять второй и третий предметы, а первый — оставить. Оба возможных решения, включающих в себя первый предмет, не являются оптимальными.

Однако для аналогичной континуальной задачи жадная стратегия, при которой сначала загружается первый товар, позволяет получить оптимальное решение, что продемонстрировано на рис. 16.2, (в). Если же сначала загрузить первый предмет в дискретной задаче, то невозможно будет загрузить рюкзак до отказа и оставшееся пустое место приведет к снижению эффективной стоимости единицы веса при такой загрузке. Принимая в дискретной задаче решение о том, следует ли помещать тот или иной предмет в рюкзак, необходимо сравнить решение подзадачи, в которую входит этот предмет, с решением подзадачи, в которой он отсутствует, и только после этого можно делать выбор. В сформулированной таким образом задаче возникает множество перекрывающихся подзадач, что служит признаком применимости динамического программирования. И в самом деле, целочисленную задачу о рюкзаке можно решить с помощью динамического программирования (см. упр. 16.2.2).

Упражнения

16.2.1

Докажите, что континуальная задача о рюкзаке обладает свойством жадного выбора.

16.2.2

Разработайте решение дискретной задачи о рюкзаке с помощью динамического программирования. Время работы вашего алгоритма должно быть равно $O(n\,W)$, где n — количество предметов, а W — максимальный вес предметов, которые вор может положить в свой рюкзак.

16.2.3

Предположим, что в дискретной задаче о рюкзаке порядок сортировки по увеличению веса совпадает с порядком сортировки по уменьшению стоимости. Сформулируйте эффективный алгоритм для поиска оптимального решения этой разновидности задачи о рюкзаке и обоснуйте его корректность.

16.2.4

Профессор едет из Киева в Москву на автомобиле. У него есть карта, на которой обозначены все заправки и указаны расстояния между ними. Известно, что если топливный бак заполнен, то автомобиль может проехать n километров (дорога достаточно ровная, так что расход топлива везде одинаков). Профессору хочется как можно реже останавливаться на заправках. Сформулируйте эффективный метод, позволяющий профессору определить, на каких заправках следует заливать топливо, чтобы количество остановок было минимальным. Докажите, что разработанная стратегия дает оптимальное решение, и определите время работы вашего алгоритма.

16.2.5

Разработайте эффективный алгоритм, который по заданному множеству $\{x_1, x_2, \ldots, x_n\}$ действительных точек на числовой прямой позволил бы найти минимальное множество закрытых интервалов единичной длины, содержащих все эти точки. Обоснуйте корректность алгоритма.

16.2.6 ★

Покажите, как решить непрерывную задачу о рюкзаке за время $O(n)$.

16.2.7

Предположим, что имеется два множества, A и B, каждое из которых состоит из n положительных целых чисел. Порядок элементов каждого множества можно менять произвольным образом. Пусть a_i — i-й элемент множества A, а b_i — i-й элемент множества B после переупорядочения. Рассмотрим величину $\prod_{i=1}^{n} {a_i}^{b_i}$. Сформулируйте алгоритм, который бы максимизировал эту величину. Докажите, что ваш алгоритм действительно максимизирует указанную величину, и определите время его работы.

16.3. Коды Хаффмана

Коды Хаффмана (Huffman codes) — очень эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20 до 90% объема. Мы рассматриваем данные, представляющие собой последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или иных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки.

Предположим, что имеется файл данных, состоящий из 100 тысяч символов, который требуется сжать. Символы в этом файле встречаются с частотой, представленной на рис. 16.3. Таким образом, всего файл содержит шесть различных символов, а, например, символ `a` встречается в нем 45 тысяч раз.

	a	b	c	d	e	f
Частота, тысяч	45	13	12	16	9	5
Кодовое слово фиксированной длины	000	001	010	011	100	101
Кодовое слово переменной длины	0	101	100	111	1101	1100

Рис. 16.3. Задача о кодировании последовательности символов. Файл данных содержит только символы `a`–`f` с указанными частотами. Если назначить каждому символу трехбитовое кодовое слово, файл можно закодировать с помощью 300 тысяч битов. При использовании показанных кодовых слов переменной длины файл кодируется только 224 тысячами битов.

Существует множество способов представить подобный файл данных. Рассмотрим задачу по разработке ***бинарного символьного кода*** (binary character code; или для краткости — просто ***код***), в котором каждый символ представляется уникальной бинарной строкой. Если используется ***код фиксированной длины***, или ***равномерный код*** (fixed-length code), в котором каждый символ представлен уникальной бинарной строкой, то для представления шести символов понадобится три бита: $a = 000$, $b = 001$, $\ldots$, $f = 101$. При использовании такого метода для кодирования всего файла понадобится 300 тысяч битов. Можно ли добиться лучших результатов?

С помощью ***кода переменной длины***, или ***неравномерного кода*** (variable-length code), удается получить значительно лучшие результаты, чем с помощью кода фиксированной длины. Это достигается за счет того, что часто встречающимся символам сопоставляются короткие кодовые слова, а редко встречающимся — длинные. Такой код представлен на рис. 16.3. В нем символ `a` представлен 1-битовой строкой 0, а символ `f` — 4-битовой строкой 1100. Для представления файла с помощью этого кода потребуется

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000 \text{ битов.}$$

Благодаря этому экономится 25% объема. Кстати, как мы вскоре убедимся, для рассматриваемого файла это оптимальная кодировка символов.

Префиксные коды

Здесь мы рассматриваем только те коды, в которых никакое кодовое слово не является префиксом какого-то другого кодового слова. Такие коды называются ***префиксными*** (prefix codes)[3]. Можно доказать (хотя здесь мы не станем этого делать), что оптимальное сжатие данных, которого можно достичь с помощью кодов, всегда может быть достигнуто при использовании префиксного кода, поэтому рассмотрение одних лишь префиксных кодов не приводит к потере общности.

Для любого бинарного кода символов кодирование текста — очень простой процесс: нужно просто соединить кодовые слова, представляющие каждый символ в файле. Например, в кодировке с помощью префиксного кода переменной длины, представленного на рис. 16.3, трехсимвольный файл abc имеет вид $0 \cdot 101 \cdot 100 = 0101100$, где символом "·" обозначена операция конкатенации.

Предпочтение префиксным кодам отдается из-за того, что они упрощают декодирование. Поскольку никакое кодовое слово не выступает в роли префикса другого, кодовое слово, с которого начинается закодированный файл, определяется однозначно. Начальное кодовое слово легко идентифицировать, преобразовать в исходный символ и продолжить декодирование оставшейся части закодированного файла. В рассматриваемом примере строка 001011101 однозначно раскладывается на подстроки $0 \cdot 0 \cdot 101 \cdot 1101$, что декодируется как aabe.

Для упрощения процесса декодирования требуется удобное представление префиксного кода, чтобы можно было легко идентифицировать начальное кодовое слово. Одним из таких представлений является бинарное дерево, листьями которого являются кодируемые символы. Бинарное кодовое слово, представляющее символ, интерпретируется как простой путь от корня к этому символу. В такой интерпретации 0 означает "перейти к левому дочернему узлу", а 1 — "перейти к правому дочернему узлу". На рис. 16.4 показаны такие деревья для двух кодов, взятых из нашего примера. Заметим, что изображенные на рисунке деревья не являются бинарными деревьями поиска, поскольку листья в них не обязательно расположены в порядке сортировки, а внутренние узлы не содержат ключей символов.

Оптимальный код файла всегда представлен *полным* бинарным деревом, каждый узел (кроме листьев) которого имеет по два дочерних узла (см. упр. 16.3.2). Код фиксированной длины, представленный в рассмотренном примере, не является оптимальным, поскольку соответствующее ему дерево, изображенное на рис. 16.4, (а), — неполное бинарное дерево: некоторые слова кода начинаются с 10..., но ни одно из них не начинается с 11.... Поскольку здесь мы можем ограничиться только полными бинарными деревьями, можно утверждать, что если C — алфавит, из которого извлекаются кодируемые символы, и все частоты, с которыми встречаются символы, положительны, то дерево, представляющее оптимальный префиксный код, содержит ровно $|C|$ листьев, по одному для каждого символа из множества C, и ровно $|C| - 1$ внутренних узлов (см. упр. Б.5.3).

---

[3] Возможно, лучше подошло бы название "беспрефиксные коды", однако "префиксные коды" — стандартный в литературе термин.

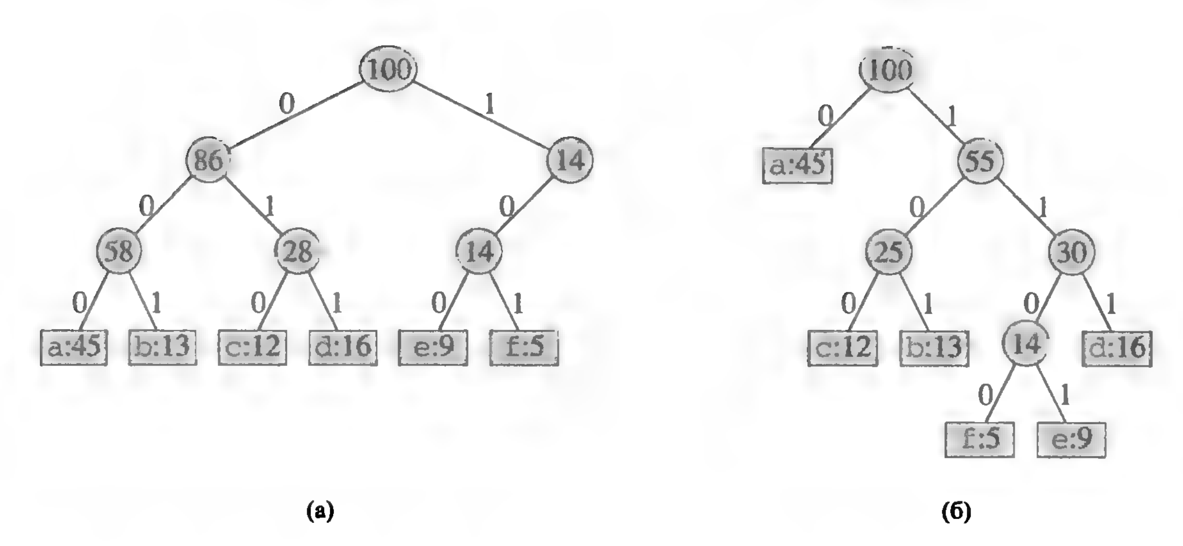


Рис. 16.4. Деревья, соответствующие схемам кодирования, представленным на рис. 16.3. Каждый лист на рисунке помечен соответствующим ему символом и частотой появления, а внутренний узел — суммой частот листьев его поддерева. **(а)** Дерево, соответствующее коду фиксированной длины a = 000, …, f = 101. **(б)** Дерево, соответствующее оптимальному префиксному коду a = 0, b = 101, …, f = 1100.

Если имеется дерево T, соответствующее префиксному коду, легко подсчитать количество битов, которые потребуются для кодирования файла. Пусть для каждого символа c в алфавите C атрибут $c.freq$ обозначает частоту появления c в файле, а $d_T(c)$ означает глубину листа, представляющего этот символ в дереве. Заметим, что $d_T(c)$ является также длиной слова, кодирующего символ c. Таким образом, для кодировки файла необходимо количество битов, равное

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) . \tag{16.4}$$

Назовем эту величину ***стоимостью*** дерева T.

Построение кода Хаффмана

Хаффман (Huffman) изобрел жадный алгоритм, позволяющий составить оптимальный префиксный код, который получил название ***код Хаффмана***. В соответствии с линией, намеченной в разделе 16.2, доказательство корректности этого алгоритма основывается на свойстве жадного выбора и оптимальной подструктуре. Вместо того чтобы демонстрировать, что эти свойства выполняются, а затем разрабатывать псевдокод, сначала представим псевдокод. Это поможет прояснить, как алгоритм осуществляет жадный выбор.

В приведенном ниже псевдокоде предполагается, что C — множество, состоящее из n символов, и что каждый из символов $c \in C$ представляет собой объект с атрибутом $c.freq$, определяющим частоту его появления. В алгоритме строится дерево T, соответствующее оптимальному коду, причем построение идет в восходящем направлении. Процесс построения начинается с множества, состоящего из $|C|$ листьев, после чего последовательно выполняется $|C| - 1$ операций “слияния”, в результате которых образуется окончательное дерево. Для идентификации двух наименее часто встречающихся объектов, подлежащих слиянию, исполь-

зуется очередь с приоритетами Q, ключами в которой являются атрибуты *freq*. В результате слияния двух объектов образуется новый объект, частота появления которого является суммой частот объединенных объектов.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      Выделение памяти для нового узла z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)        // Возврат корня дерева
```

Для рассмотренного ранее примера алгоритм Хаффмана работает так, как показано на рис. 16.5. Поскольку алфавит содержит шесть букв, начальный размер очереди равен $n = 6$, и дерево строится за пять шагов слияния. Полученное в результате дерево представляет оптимальный префиксный код. Кодовое слово буквы представляет собой последовательность меток ребер на простом пути от корня к листу с этой буквой.

В строке 2 инициализируется неубывающая очередь с приоритетами Q, состоящая из элементов множества C. Цикл **for** в строках 3–8 поочередно извлекает по два узла, x и y, которые характеризуются в очереди наименьшими частотами, и заменяет их в очереди новым узлом z, представляющим объединение упомянутых выше элементов. Частота появления z вычисляется в строке 7 как сумма частот x и y. Узел x является левым дочерним узлом z, а y — его правым дочерним узлом. (Этот порядок является произвольным; перестановка левого и правого дочерних узлов приводит к созданию другого кода с той же стоимостью.) После $n - 1$ объединений в очереди остается один узел — корень дерева кодов, который возвращается в строке 9.

Хотя алгоритм приводил бы к такому же результату при полном исключении переменных x и y — путем непосредственного присваивания соответствующих значений атрибутам $z.left$ и $z.right$ в строках 5 и 6 и замены строки 7 на $z.freq = z.left.freq + z.right.freq$, — мы все же будем использовать имена x и y в доказательстве корректности алгоритма. Поэтому лучше оставить эти переменные в псевдокоде.

В процессе анализа времени работы алгоритма Хаффмана предполагается, что Q реализована как бинарная неубывающая пирамида (см. главу 6). Для множества C, состоящего из n символов, инициализацию очереди Q в строке 2 можно выполнить за время $O(n)$ с помощью процедуры BUILD-MIN-HEAP из раздела 6.3. Цикл **for** в строках 3–8 выполняется ровно $n - 1$ раз, а поскольку для каждой операции над пирамидой требуется время $O(\lg n)$, вклад цикла во время работы алгоритма равен $O(n \lg n)$. Таким образом, полное время работы процедуры HUFFMAN с входным множеством, состоящим из n символов, равно $O(n \lg n)$.

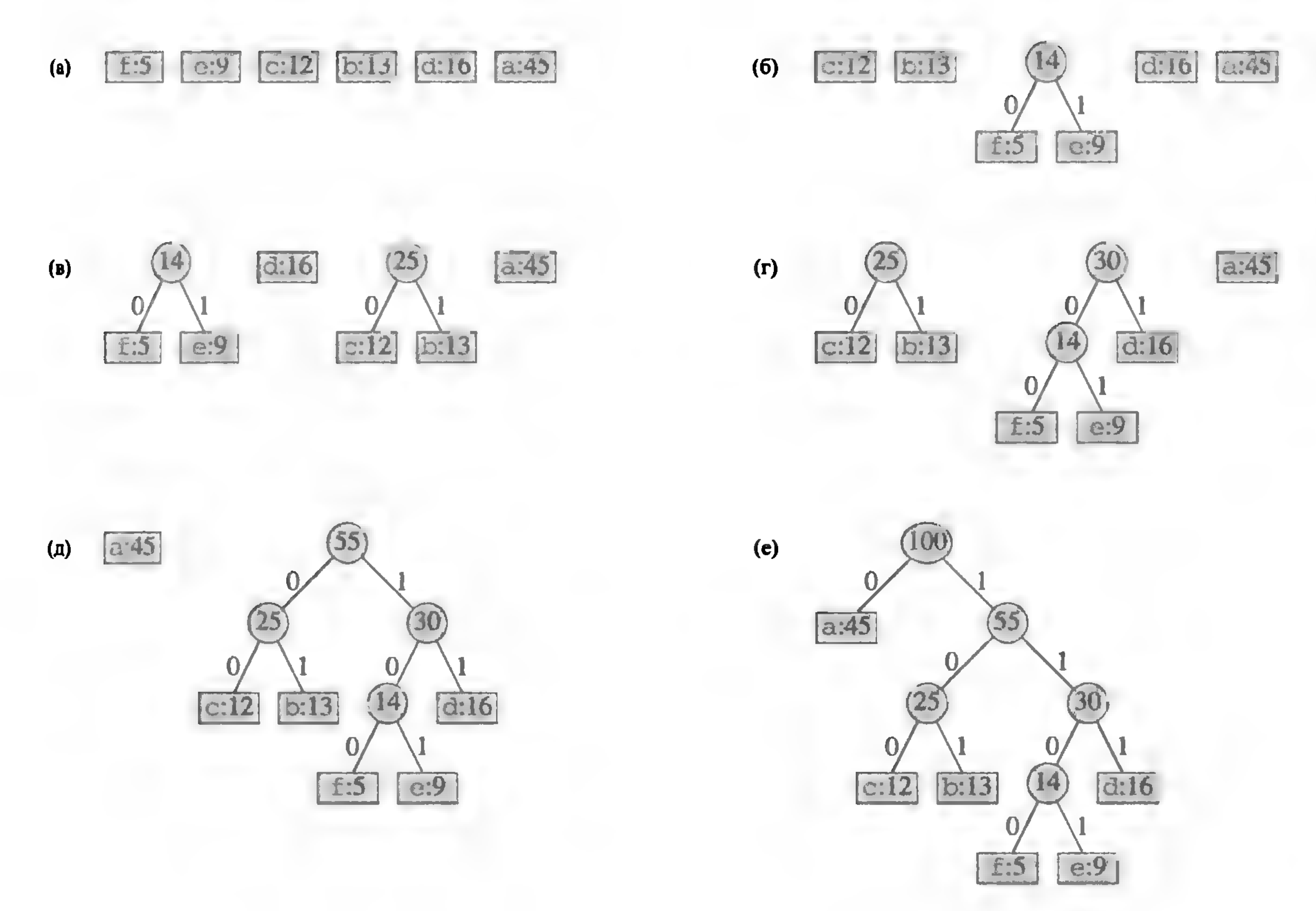


Рис. 16.5. Этапы работы алгоритма Хаффмана для частот, заданных на рис. 16.3. В каждой части показано содержимое очереди, элементы которой рассортированы в порядке возрастания их частот. На каждом шаге работы алгоритма объединяются два объекта (дерева) с самыми низкими частотами. Листья изображены в виде прямоугольников, в каждом из которых указаны буква и ее частота. Внутренние узлы представлены кругами, содержащими сумму частот дочерних узлов. Ребро, соединяющее внутренний узел с левым дочерним узлом, имеет метку 0, а ребро, соединяющее его с правым дочерним узлом, — метку 1. Кодовое слово для буквы образуется последовательностью меток на ребрах, соединяющих корень с листом, представляющим эту букву. **(а)** Начальное множество из $n = 6$ узлов, по одному для каждой буквы. **(б)–(д)** Промежуточные шаги. **(е)** Окончательное дерево.

Это время можно сократить до $O(n \lg \lg n)$, если вместо неубывающей пирамиды воспользоваться деревом ван Эмде Боаса (см. главу 20).

Корректность алгоритма Хаффмана

Чтобы доказать корректность жадного алгоритма HUFFMAN, покажем, что задача о построении оптимального префиксного кода демонстрирует свойства жадного выбора и оптимальной подструктуры. В сформулированной ниже лемме показано соблюдение свойства жадного выбора.

Лемма 16.2

Пусть C — алфавит, каждый символ $c \in C$ которого встречается с частотой $c.freq$. Пусть x и y — два символа алфавита C с самыми низкими частотами. Тогда для алфавита C существует оптимальный префиксный код, кодовые слова символов x и y в котором имеют одинаковую длину и отличаются лишь последним битом.

Доказательство. Идея доказательства состоит в том, чтобы взять дерево T, представляющее произвольный оптимальный префиксный код, и преобразовать его в дерево, представляющее другой оптимальный префиксный код, в котором символы x и y являются листьями с общим родительским узлом, причем в новом дереве эти листья находятся на максимальной глубине. Если мы сможем построить такое дерево, то кодовые слова для x и y будут иметь одинаковую длину и отличаться только последним битом.

Пусть a и b — два символа, представленные листьями с общим родительским узлом на максимальной глубине в T. Без потери общности считаем, что $a.\mathit{freq} \leq b.\mathit{freq}$ и $x.\mathit{freq} \leq y.\mathit{freq}$. Поскольку $x.\mathit{freq}$ и $y.\mathit{freq}$ — две самые маленькие частоты (в указанном порядке), а $a.\mathit{freq}$ и $b.\mathit{freq}$ — две произвольные частоты, мы имеем $x.\mathit{freq} \leq a.\mathit{freq}$ и $y.\mathit{freq} \leq b.\mathit{freq}$.

В оставшейся части доказательства возможно выполнение равенства $x.\mathit{freq} = a.\mathit{freq}$ или $y.\mathit{freq} = b.\mathit{freq}$. Однако если $x.\mathit{freq} = b.\mathit{freq}$, то должно выполняться и $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$ (см. упр. 16.3.1), и лемма тривиально истинна. Таким образом, мы будем считать, что $x.\mathit{freq} \neq b.\mathit{freq}$, что означает, что $x \neq b$.

Как показано на рис. 16.6, в результате перестановки в дереве T листьев a и x получается дерево T', а при последующей перестановке в дереве T' листьев b и y получается дерево T'', в котором x и y — листья-братья на максимальной глубине. (Заметим, что если $x = b$, но $y \neq a$, то x и y в дереве T'' не являются листьями-братьями на максимальной глубине. Поскольку мы считаем, что $x \neq b$, такая ситуация возникнуть не может.)

Согласно уравнению (16.4) разность стоимостей T и T' равна

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&\quad = (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\quad \geq 0\ ,
\end{aligned}$$

поскольку и $a.\mathit{freq} - x.\mathit{freq}$, и $d_T(a) - d_T(x)$ неотрицательны. Точнее говоря, величина $a.\mathit{freq} - x.\mathit{freq}$ неотрицательна, поскольку x является листом с минимальной частотой, а $d_T(a) - d_T(x)$ неотрицательна, поскольку a представляет собой лист максимальной глубины в T. Аналогично обмен y и b не увеличивает стоимости, так что значение $B(T') - B(T'')$ неотрицательное. Следовательно, $B(T'') \leq B(T)$, а поскольку T — оптимальное дерево, мы имеем $B(T) \leq B(T'')$, откуда вытекает $B(T'') = B(T)$. Таким образом, T'' является оптимальным деревом, в котором x и y представляют собой листья-братья на максимальной глубине, что и доказывает лемму. ■

Из леммы 16.2 следует, что процесс построения оптимального дерева путем объединения узлов без потери общности можно начать с жадного выбора, при

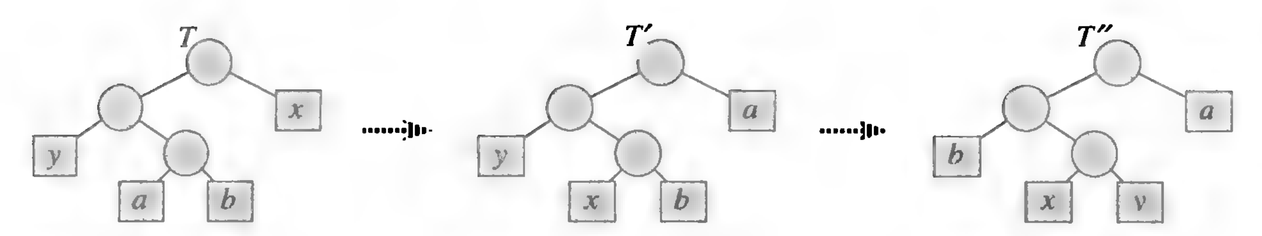


Рис. 16.6. Иллюстрация ключевых этапов доказательства леммы 16.2. В оптимальном дереве T листья a и b являются двумя братьями на максимальной глубине. Листья x и y представляют собой два символа с наименьшими частотами; они располагаются в произвольных позициях в T. В предположении, что $x \neq b$, обмен листьев a и x дает дерево T', после чего обмен листьев b и y дает дерево T''. Поскольку каждый обмен не увеличивает стоимость, полученное дерево T'' также является оптимальным.

котором объединению подлежат два символа с наименьшими частотами. Почему такой выбор будет жадным? Стоимость объединения можно рассматривать как сумму частот входящих в него элементов. В упр. 16.3.4 предлагается показать, что полная стоимость сконструированного таким образом дерева равна сумме стоимостей его составляющих. Из всевозможных вариантов объединения на каждом этапе в процедуре HUFFMAN выбирается тот, в котором получается минимальная стоимость.

В приведенной ниже лемме показано, что задача о составлении оптимальных префиксных кодов обладает свойством оптимальной подструктуры.

Лемма 16.3

Пусть имеется алфавит C, в котором для каждого символа $c \in C$ определены частоты $c.\mathit{freq}$. Пусть x и y — два символа из алфавита C с минимальными частотами. Пусть C' — алфавит, полученный из алфавита C путем удаления символов x и y и добавления нового символа z, так что $C' = C - \{x, y\} \cup \{z\}$. Определим частоты freq в C', как и в C, за исключением того, что $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Пусть T' — произвольное дерево, представляющее оптимальный префиксный код для алфавита C'. Тогда дерево T, полученное из дерева T' путем замены листа z внутренним узлом с дочерними элементами x и y, представляет оптимальный префиксный код для алфавита C.

Доказательство. Сначала покажем, как выразить стоимость $B(T)$ дерева T через стоимость $B(T')$ дерева T', рассматривая стоимость компонентов в уравнении (16.4). Для каждого символа $c \in C - \{x, y\}$ мы имеем $d_T(c) = d_{T'}(c)$, и, следовательно, $c.\mathit{freq} \cdot d_T(c) = c.\mathit{freq} \cdot d_{T'}(c)$. Поскольку $d_T(x) = d_T(y) = d_{T'}(z) + 1$, получаем

$$\begin{aligned} x.\mathit{freq} \cdot d_T(x) + y.\mathit{freq} \cdot d_T(y) &= (x.\mathit{freq} + y.\mathit{freq})(d_{T'}(z) + 1) \\ &= z.\mathit{freq} \cdot d_{T'}(z) + (x.\mathit{freq} + y.\mathit{freq}) \ , \end{aligned}$$

откуда заключаем, что

$$B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$$

или, что то же самое,

$$B(T') = B(T) - x.freq - y.freq \ .$$

Докажем лемму методом от противного. Предположим, дерево T не представляет оптимальный префиксный код для алфавита C. Тогда существует дерево T'', для которого справедливо неравенство $B(T'') < B(T)$. Согласно лемме 16.2 x и y без потери общности можно считать дочерними элементами одного и того же узла в T''. Пусть дерево T''' получено из дерева T'' путем замены родителя x и y листом z с частотой $z.freq = x.freq + y.freq$. Тогда можно записать

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \ , \end{aligned}$$

что противоречит предположению о том, что дерево T' представляет оптимальный префиксный код для алфавита C'. Таким образом, дерево T должно представлять оптимальный префиксный код для алфавита C. ■

Теорема 16.4

Процедура HUFFMAN дает оптимальный префиксный код.

Доказательство. Справедливость теоремы непосредственно следует из лемм 16.2 и 16.3. ■

Упражнения

16.3.1

Поясните, почему в доказательстве леммы 16.2 при $x.freq = b.freq$ должно выполняться соотношение $a.freq = b.freq = x.freq = y.freq$.

16.3.2

Докажите, что бинарное дерево, не являющееся полным, не может соответствовать оптимальному префиксному коду.

16.3.3

Что собой представляет оптимальный код Хаффмана для представленного ниже множества частот, основанного на первых восьми числах Фибоначчи.

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Попытайтесь обобщить ответ на случай первых n чисел Фибоначчи.

16.3.4

Докажите, что полную стоимость дерева, представляющего какой-нибудь код, можно также вычислить как сумму частот двух дочерних узлов по всем внутренним узлам.

16.3.5

Докажите, что если символы отсортированы в монотонно невозрастающем порядке частот, то существует оптимальный код, длины слов в котором представляют собой монотонно неубывающие величины.

16.3.6

Предположим, имеется оптимальный префиксный код для множества символов $C = \{0, 1, \ldots, n-1\}$, и мы хотим передать его, использовав как можно меньше битов. Покажите, как представить любой оптимальный префиксный код для множества C с помощью всего $2n - 1 + n\lceil\lg n\rceil$ бит. (*Указание:* для представления структуры дерева, определяемой его обходом, достаточно $2n - 1$ бит.)

16.3.7

Обобщите алгоритм Хаффмана для кодовых слов в троичной системе счисления (т.е. для слов, в которых используются символы 0, 1 и 2) и докажите, что с его помощью получается оптимальный троичный код.

16.3.8

Предположим, что файл данных содержит последовательность 8-битовых символов, причем все 256 символов встречаются почти одинаково часто: максимальная частота превосходит минимальную менее чем в два раза. Докажите, что в этом случае кодирование Хаффмана по эффективности не превышает обычный 8-битовый код фиксированной длины.

16.3.9

Покажите, что в среднем ни одна схема сжатия не в состоянии даже на один бит сжать файл, состоящий из случайных 8-битовых символов. (*Указание:* сравните количество возможных файлов с количеством сжатых файлов.)

⋆ 16.4. Матроиды и жадные методы

В этом разделе в общих чертах рассматривается элегантная теория жадных алгоритмов. Она оказывается полезной, когда нужно определить, когда жадный метод дает оптимальное решение. Эта теория содержит комбинаторные структуры, известные под названием "матроиды". Несмотря на то что рассматриваемая теория не описывает все случаи, в которых применим жадный метод (например, она не описывает задачу о выборе процессов из раздела 16.1 или задачу о кодах Хаффмана из раздела 16.3), она находит применение во многих случаях, представляющих практический интерес. Более того, эта теория быстро развивается и обобщается, находя все больше сфер приложения (см. в конце данной главы ссылки на литературу по этой теме).

Матроиды

Матроид (matroid) — это упорядоченная пара $M = (S, \mathcal{I})$, удовлетворяющая сформулированным ниже условиям.

1. S является конечным множеством.
2. $\mathcal{I}$ — непустое семейство подмножеств множества S (которые называются ***независимыми*** подмножествами), таких, что если $B \in \mathcal{I}$ и $A \subseteq B$, то $A \in \mathcal{I}$. Если семейство $\mathcal{I}$ удовлетворяет этому свойству, то его называют ***наследственным*** (hereditary). Заметим, что пустое множество $\emptyset$ с необходимостью принадлежит семейству $\mathcal{I}$.
3. Если $A \in \mathcal{I}$, $B \in \mathcal{I}$ и $|A| < |B|$, то существует такой элемент $x \in B - A$, что $A \cup \{x\} \in \mathcal{I}$. Говорят, что структура M удовлетворяет ***свойству замены*** (exchange property).

Термин "матроид" был введен Хасслером Уитни (Hassler Whitney). Он изучал ***матричные матроиды***, элементами S в которых являются строки заданной матрицы. Множество строк является независимым, если они линейно независимы в обычном смысле. Легко показать, что эта структура определяет матроид (см. упр. 16.4.2).

В качестве другого примера матроида рассмотрим ***графовый матроид*** $M_G = (S_G, \mathcal{I}_G)$, определенный ниже в терминах неориентированного графа $G = (V, E)$ следующим образом.

- Множество S_G представляет собой множество E ребер графа G.
- Если A является подмножеством множества E, то $A \in \mathcal{I}_G$ тогда и только тогда, когда множество A ациклическое, т.е. множество ребер A является независимым тогда и только тогда, когда подграф $G_A = (V, A)$ образует лес.

Графовый матроид M_G тесно связан с задачей о минимальном остовном дереве, подробно описанной в главе 23.

Теорема 16.5

Если $G = (V, E)$ представляет собой неориентированный граф, то $M_G = (S_G, \mathcal{I}_G)$ является матроидом.

Доказательство. Очевидно, что $S_G = E$ — конечное множество. Кроме того, $\mathcal{I}_G$ — наследственное семейство, поскольку подмножество леса является лесом. Другими словами, удаление ребер из ациклического множества ребер не может привести к образованию циклов.

Таким образом, осталось показать, что структура M_G удовлетворяет свойству обмена. Предположим, что $G_A = (V, A)$ и $G_B = (V, B)$ — леса графа G и что $|B| > |A|$, т.е. A и B — ациклические множества ребер, и в множестве B содержится больше ребер, чем в множестве A.

Мы утверждаем, что лес $F = (V_F, E_F)$ содержит ровно $|V_F| - |E_F|$ деревьев. Чтобы понять, почему это так, предположим, что F состоит из t деревьев, где i-е

дерево содержит v_i вершин и e_i ребер. Тогда мы имеем

$$\begin{aligned} |E_F| &= \sum_{i=1}^{t} e_i \\ &= \sum_{i=1}^{t} (v_i - 1) && \text{(согласно теореме Б.2)} \\ &= \sum_{i=1}^{t} v_i - t \\ &= |V_F| - t \;, \end{aligned}$$

откуда вытекает, что $t = |V_F| - |E_F|$. Таким образом, лес G_A содержит $|V| - |A|$ деревьев, а лес G_B содержит $|V| - |B|$ деревьев.

Поскольку в лесу G_B меньше деревьев, чем в лесу G_A, в G_B должно содержаться некоторое дерево T, вершины которого принадлежат двум различным деревьям леса G_A. Более того, так как дерево T — связное, оно должно содержать такое ребро (u, v), что вершины u и v принадлежат разным деревьям леса G_A. Поскольку ребро (u, v) соединяет вершины двух разных деревьев леса G_A, его можно добавить в лес G_A, не образовав при этом цикла. Таким образом, структура M_G удовлетворяет свойству обмена, что и завершает доказательство того, что M_G является матроидом. ■

Для заданного матроида $M = (S, \mathcal{I})$ назовем элемент $x \notin A$ ***расширением*** (extension) множества $A \in \mathcal{I}$, если его можно добавить в A без нарушения независимости, т.е. x является расширением множества A, если $A \cup \{x\} \in \mathcal{I}$. В качестве примера рассмотрим графовый матроид M_G. Если A — независимое множество ребер, то ребро e является расширением множества A тогда и только тогда, когда оно не принадлежит этому множеству и его добавление в A не приводит к образованию цикла.

Если A — независимое подмножество в матроиде M, то, если у него нет расширений, говорят, что A — ***максимальное*** множество. Таким образом, множество A является максимальным, если оно не содержится ни в одном большем независимом подмножестве матроида M. Сформулированное ниже свойство часто оказывается весьма полезным.

Теорема 16.6

Все максимальные независимые подмножества матроида имеют один и тот же размер.

Доказательство. Докажем теорему методом от противного. Предположим, что A — максимальное независимое подмножество матроида M и что существует другое максимальное независимое подмножество B матроида M, размер которого превышает размер подмножества A. Тогда из свойства обмена следует, что множество A расширяемо до большего независимого множества $A \cup \{x\}$ за счет

некоторого элемента $x \in B - A$, что противоречит предположению о максимальности множества A. ■

В качестве иллюстрации применения этой теоремы рассмотрим графовый матроид M_G связного неориентированного графа G. Каждое максимальное независимое подмножество M_G должно представлять собой свободное дерево, содержащее ровно $|V| - 1$ ребер, которые соединяют все вершины графа G. Такое дерево называется ***остовным деревом*** (spanning tree) графа G.

Говорят, что матроид $M = (S, \mathcal{I})$ ***взвешенный*** (weighted), если с ним связана весовая функция w, назначающая каждому элементу $x \in S$ строго положительный вес $w(x)$. Весовая функция w обобщается на подмножества S путем суммирования

$$w(A) = \sum_{x \in A} w(x)$$

для любого подмножества $A \subseteq S$. Например, если обозначить вес ребра e графового матроида M_G через $w(e)$, то $w(A)$ представляет собой суммарный вес всех ребер, принадлежащих множеству A.

Жадные алгоритмы на взвешенном матроиде

Многие задачи, для которых жадный подход позволяет получить оптимальное решение, можно сформулировать в терминах поиска независимого подмножества с максимальным весом во взвешенном матроиде. То есть задан взвешенный матроид $M = (S, \mathcal{I})$, и нужно найти независимое множество $A \in \mathcal{I}$, для которого величина $w(A)$ будет максимальной. Назовем такое максимальное независимое подмножество с максимально возможным весом ***оптимальным*** подмножеством матроида. Поскольку вес $w(x)$ любого элемента $x \in S$ положителен, оптимальное подмножество всегда является максимальным независимым подмножеством, что всегда помогает сделать множество A бóльшим, насколько это возможно.

Например, в ***задаче о минимальном остовном дереве*** (minimum-spanning-tree problem) задаются связный неориентированный граф $G = (V, E)$ и функция длин w, такая, что $w(e)$ — (положительная) длина ребра e. (Термин "длина" используется здесь для обозначения исходного веса, соответствующего ребру графа; термин "вес" сохранен для обозначения весов в соответствующем матроиде.) Необходимо найти подмножество ребер, которые соединяют все вершины и имеют минимальную общую длину. Чтобы представить эту проблему в виде задачи поиска оптимального подмножества матроида, рассмотрим взвешенный матроид M_G с весовой функцией w', где $w'(e) = w_0 - w(e)$, а величина w_0 превышает максимальную длину любого ребра. В таком взвешенном матроиде любой вес является положительным, а оптимальное подмножество представляет собой остовное дерево в исходном графе, имеющее минимальную общую длину. Точнее говоря, каждое максимальное независимое подмножество A соответствует остовному дереву с $|V| - 1$ ребрами, и поскольку для любого максимального независимого

подмножества A справедливо соотношение

$$\begin{aligned}
w'(A) &= \sum_{e \in A} w'(e) \\
&= \sum_{e \in A} (w_0 - w(e)) \\
&= (|V| - 1)w_0 - \sum_{e \in A} w(e) \\
&= (|V| - 1)w_0 - w(A) ,
\end{aligned}$$

независимое подмножество, которое максимизирует величину $w'(A)$, должно минимизировать величину $w(A)$. Таким образом, любой алгоритм, позволяющий найти оптимальное подмножество A произвольного матроида, позволяет также решить задачу о минимальном остовном дереве.

В главе 23 приводится алгоритм решения задачи о минимальном остовном дереве, а здесь описывается жадный алгоритм, который работает для произвольного взвешенного матроида. В качестве входных данных в этом алгоритме выступают взвешенный матроид $M = (S, \mathcal{I})$ и связанная с ним весовая функция w. Алгоритм возвращает оптимальное подмножество A. В рассматриваемом псевдокоде компоненты матроида M обозначены как $M.S$ и $M.\mathcal{I}$, а весовая функция — как w. Алгоритм является жадным, поскольку все элементы $x \in S$ рассматриваются в нем в порядке монотонного убывания веса, причем элемент тут же добавляется в множество A, если множество $A \cup \{x\}$ является независимым.

GREEDY(M, w)

1 $A = \emptyset$
2 Отсортировать $M.S$ в невозрастающем порядке весов w
3 **for** Каждый $x \in M.S$ в невозрастающем порядке весов $w(x)$
4   **if** $A \cup \{x\} \in M.\mathcal{I}$
5         $A = A \cup \{x\}$
6 **return** A

В строке 4 для каждого элемента x проверяется, останется ли A после его добавления независимым множеством. Если останется, в строке 5 выполняется добавление x в A. В противном случае x отвергается. Поскольку пустое множество является независимым и поскольку каждая итерация цикла **for** поддерживает независимость A, подмножество A всегда независимо по индукции. Следовательно, процедура GREEDY всегда возвращает независимое подмножество A. Вскоре мы увидим, что A является подмножеством с максимальным возможным весом, так что A представляет собой оптимальное подмножество.

Время работы процедуры GREEDY легко проанализировать. Обозначим через n величину $|S|$. Фаза сортировки длится в течение времени $O(n \lg n)$. Строка 4 выполняется ровно n раз, по одному разу для каждого элемента множества S. При каждом выполнении строки 4 требуется проверить, является ли независимым множество $A \cup \{x\}$. Если каждая подобная проверка длится в течение времени $O(f(n))$, общее время работы алгоритма составляет $O(n \lg n + nf(n))$.

Теперь докажем, что процедура GREEDY возвращает оптимальное подмножество.

Лемма 16.7 (Матроиды обладают свойством жадного выбора)

Предположим, что $M = (S, \mathcal{I})$ — взвешенный матроид с весовой функцией w и что множество S отсортировано в невозрастающем порядке весов. Пусть x — первый элемент множества S, такой, что множество $\{x\}$ независимо (если такой x существует). Если элемент x существует, то существует и оптимальное подмножество A множества S, содержащее элемент x.

Доказательство. Если такого элемента x не существует, то единственным независимым подмножеством является пустое множество и доказательство закончено. В противном случае предположим, что B — произвольное непустое оптимальное подмножество. Предположим также, что $x \notin B$; в противном случае считаем, что $A = B$ дает оптимальное подмножество S, которое содержит x.

Ни один из элементов множества B не имеет вес, больший, чем $w(x)$. Чтобы увидеть, почему это так, заметим, что из $y \in B$ следует, что множество $\{y\}$ независимо, поскольку $B \in \mathcal{I}$, а семейство $\mathcal{I}$ наследственное. Таким образом, благодаря выбору элемента x обеспечивается выполнение неравенства $w(x) \geq w(y)$ для любого элемента $y \in B$.

Построим множество A, как описано ниже. Начнем с $A = \{x\}$. В соответствии с выбором элемента x множество A — независимое. С помощью свойства обмена будем многократно осуществлять поиск нового элемента множества B, который можно добавить в множество A, пока не будет достигнуто равенство $|A| = |B|$; при этом множество A останется независимым. В этот момент A и B одинаковы, с тем отличием, что A содержит x, а B — некоторый другой элемент y. То есть $A = B - \{y\} \cup \{x\}$ для некоторого элемента $y \in B$, так что

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B) \ . \end{aligned}$$

Поскольку множество B оптимально, множество A, содержащее x, также должно быть оптимальным. ■

Теперь покажем, что если какой-то элемент не может быть добавлен изначально, то он не может быть добавлен и позже.

Лемма 16.8

Пусть $M = (S, \mathcal{I})$ — произвольный матроид. Если x — элемент S, представляющий собой расширение некоторого независимого подмножества A множества S, то x также является расширением пустого множества $\emptyset$.

Доказательство. Поскольку x — расширение множества A, множество $A \cup \{x\}$ независимое. Так как семейство $\mathcal{I}$ является наследственным, множество $\{x\}$ должно быть независимым. Таким образом, x — расширение пустого множества $\emptyset$. ■

Следствие 16.9

Пусть $M = (S, \mathcal{I})$ — произвольный матроид. Если x — элемент множества S, который не является расширением пустого множества $\emptyset$, то этот элемент также не является расширением любого независимого подмножества A множества S.

Доказательство. Это следствие — просто обращение леммы 16.8. ■

В следствии 16.9 утверждается, что любой элемент, который не может быть использован сразу, не может использоваться никогда. Таким образом, в процедуре GREEDY не может быть допущена ошибка, состоящая в пропуске какого-нибудь начального элемента из множества S, который не является расширением пустого множества $\emptyset$, поскольку такие элементы никогда не могут быть использованы.

Лемма 16.10 (Матроиды обладают свойством оптимальной подструктуры)

Пусть x — первый элемент множества S, выбранный алгоритмом GREEDY для взвешенного матроида $M = (S, \mathcal{I})$. Оставшаяся задача поиска независимого подмножества с максимальным весом, содержащего элемент x, сводится к поиску независимого подмножества с максимальным весом для взвешенного матроида $M' = (S', \mathcal{I}')$, где

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\} ,$$
$$\mathcal{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\} ,$$

и весовая функция матроида M' совпадает с весовой функцией матроида M, ограниченной множеством S'. (Назовем матроид M' ***сужением*** (contraction) матроида M на элемент x.)

Доказательство. Если A — произвольное независимое подмножество с максимальным весом матроида M, содержащее элемент x, то $A' = A - \{x\}$ представляет собой независимое подмножество матроида M'. Справедливо также обратное: из любого независимого подмножества A' матроида M' можно получить независимое подмножество $A = A' \cup \{x\}$ матроида M. Поскольку в обоих случаях выполняется соотношение $w(A) = w(A') + w(x)$, решение с максимальным весом для матроида M, содержащее элемент x, позволяет получить решение с максимальным весом для матроида M', и наоборот. ■

Теорема 16.11 (Корректность жадного алгоритма для матроидов)

Если $M = (S, \mathcal{I})$ — взвешенный матроид с весовой функцией w, то процедура GREEDY(M, w) возвращает оптимальное подмножество.

Доказательство. Согласно следствию 16.9 обо всех изначально пропущенных процедурой GREEDY элементах, не являющихся расширениями пустого множества $\emptyset$, можно забыть, поскольку они никогда больше не понадобятся. Когда выбран первый элемент x, из леммы 16.7 следует, что процедура GREEDY не допускает ошибки, добавляя элемент x в множество A, потому что существует оптимальное подмножество, содержащее элемент x. И наконец из леммы 16.10 следует, что в оставшейся задаче требуется найти оптимальное подмножество

матроида M', представляющего собой сужение матроида M на элемент x. После того как в процедуре GREEDY множество A приобретет вид $\{x\}$, все остальные действия этой процедуры можно интерпретировать как действия над матроидом $M' = (S', \mathcal{I}')$. Это утверждение справедливо благодаря тому, что любое множество $B \in \mathcal{I}'$ — независимое подмножество матроида M' тогда и только тогда, когда множество $B \cup \{x\}$ является независимым в матроиде M. Таким образом, в ходе последующей работы процедуры GREEDY будет найдено независимое подмножество с максимальным весом для матроида M', а в результате полного выполнения этой процедуры будет найдено независимое подмножество с максимальным весом для матроида M. ■

Упражнения

16.4.1

Покажите, что если S — произвольное конечное множество, а $\mathcal{I}_k$ — множество всех подмножеств S, размер которых не превышает k, где $k \leq |S|$, то $(S, \mathcal{I}_k)$ является матроидом.

16.4.2 ★

Пусть T — матрица размером $m \times n$ над некоторым полем (например, действительных чисел). Покажите, что если S — множество столбцов T, а $A \in \mathcal{I}$, то $(S, \mathcal{I})$ является матроидом тогда и только тогда, когда столбцы матрицы A линейно независимы.

16.4.3 ★

Покажите, что если $(S, \mathcal{I})$ — матроид, то матроидом является и $(S, \mathcal{I}')$, где

$$\mathcal{I}' = \{A' : S - A' \text{ содержит некоторое максимальное } A \in \mathcal{I}\} ,$$

т.е. максимальные независимые множества матроида $(S, \mathcal{I}')$ представляют собой дополнения максимальных независимых множеств матроида $(S, \mathcal{I})$.

16.4.4 ★

Пусть S — конечное множество, а $S_1, S_2, \ldots, S_k$ — разбиение этого множества на непустые непересекающиеся подмножества. Определим структуру $(S, \mathcal{I})$ с помощью условия $\mathcal{I} = \{A : |A \cap S_i| \leq 1 \text{ для } i = 1, 2, \ldots, k\}$. Покажите, что $(S, \mathcal{I})$ — матроид. Другими словами, множество всех множеств A, содержащих не более одного члена в каждом подмножестве разбиения, определяет независимые множества матроида.

16.4.5

Покажите, как можно преобразовать весовую функцию в задаче о взвешенном матроиде, в которой оптимальное решение представляет собой максимальное независимое подмножество *с минимальным весом*, чтобы преобразовать ее в стандартную задачу о взвешенном матроиде. Обоснуйте корректность преобразования.

★ 16.5. Планирование заданий как матроид

Интересная задача, которую можно решить с помощью матроидов, — составление оптимального расписания единичных заданий, выполняющихся на одном процессоре. Каждое задание характеризуется конечным сроком выполнения, а также штрафом при пропуске этого срока. Эта задача кажется сложной, однако она на удивление просто решается с помощью жадного алгоритма.

Единичное задание (unit-time job) — это задание (например, компьютерная программа), для выполнения которого требуется единичный интервал времени. Если имеется конечное множество S таких заданий, ***расписание*** (schedule) для этого множества представляет собой перестановку элементов множества S, определяющую порядок их выполнения. Первое задание в расписании начинается в нулевой момент времени и заканчивается в момент времени 1, второе задание начинается в момент времени 1 и заканчивается в момент времени 2, и т.д.

Входные данные в задаче по ***планированию на одном процессоре единичных заданий, характеризующихся конечным сроком выполнения и штрафом***, имеют следующий вид:

- множество $S = \{a_1, a_2, \ldots, a_n\}$, состоящее из n единичных заданий;
- множество $d_1, d_2, \ldots, d_n$ из n ***конечных сроков выполнения***, представленных целыми числами $1 \leq d_i \leq n$; предполагается, что задание a_i должно завершиться до момента времени d_i;
- множество из n неотрицательных весов или ***штрафных сумм*** $w_1, w_2, \ldots, w_n$; если задание a_i не будет выполнено к моменту времени d_i, изымается штраф w_i; если это задание будет выполнено в срок, штрафные санкции не применяются.

Для множества заданий S нужно найти расписание, минимизирующее суммарный штраф, который накладывается за все просроченные задания.

Рассмотрим произвольное расписание. Будем называть задание в нем ***просроченным***, если оно завершается позже конечного срока выполнения. В противном случае задание ***своевременное***. Произвольное расписание всегда можно привести к ***виду с первоочередными своевременными заданиями*** (early-first form), когда своевременные задания выполняются перед просроченными. Чтобы продемонстрировать это, заметим, что если какое-нибудь своевременное задание a_i следует после некоторого просроченного задания a_j, то задания a_i и a_j можно поменять местами, причем задание a_i все равно останется своевременным, а задание a_j — просроченным.

Кроме того, справедливо утверждение, согласно которому произвольное расписание всегда можно привести к ***каноническому виду*** (canonical form), в котором своевременные задания предшествуют просроченным и расположены в порядке неубывания конечных сроков выполнения. Для этого сначала приведем расписание к виду с первоочередными своевременными заданиями. После этого, до тех пор пока в расписании будут иметься своевременные задания a_i и a_j, которые заканчиваются в моменты времени k и $k + 1$ соответственно, но при этом

$d_j < d_i$, мы будем менять их местами. Поскольку задание a_j до перестановки было своевременным, $k+1 \leq d_j$. Таким образом, $k+1 < d_i$, и задание a_i остается своевременным и после перестановки. Задание a_j сдвигается на более раннее время, поэтому после перестановки оно также остается своевременным.

Приведенные выше рассуждения позволяют свести поиск оптимального расписания к определению множества A, состоящего из своевременных заданий в оптимальном расписании. Как только такое множество будет определено, можно будет создать фактическое расписание, включив в него элементы множества A в порядке монотонного возрастания моментов их окончания, а затем перечислив просроченные задания $(S - A)$ в произвольном порядке. Таким образом будет получено канонически упорядоченное оптимальное расписание.

Множество заданий A называется ***независимым***, если для него существует расписание, в котором отсутствуют просроченные задания. Очевидно, что множество своевременных заданий расписания образует независимое множество заданий. Обозначим через $\mathcal{I}$ семейство всех независимых множеств заданий.

Рассмотрим задачу, состоящую в определении того, является ли заданное множество заданий A независимым. Обозначим через $N_t(A)$ количество заданий в множестве A, конечный срок выполнения которых равен t или наступает раньше (величина t может принимать значения $0, 1, 2, \ldots, n$). Заметим, что $N_0(A) = 0$ для любого множества A.

Лемма 16.12

Для любого множества заданий A сформулированные ниже утверждения эквивалентны.

1. Множество A независимое.
2. Неравенство $N_t(A) \leq t$ справедливо для всех $t = 0, 1, 2, \ldots, n$.
3. Если в расписании задания из множества A расположены в порядке неубывания конечных сроков выполнения, то ни одно из них не является просроченным.

Доказательство. Чтобы показать, что из (1) следует (2), докажем обращенное утверждение: если $N_t(A) > t$ для некоторого t, то невозможно составить расписание таким образом, чтобы в множестве A не оказалось просроченных заданий, поскольку до наступления момента t остается более t незавершенных заданий. Таким образом, утверждение (1) предполагает выполнение утверждения (2). Если выполняется утверждение (2), то i-й по порядку срок завершения задания не превышает i, так что при расстановке заданий в этом порядке все сроки будут соблюдены. Наконец из утверждения (3) тривиальным образом следует справедливость утверждения (1). ■

С помощью свойства 2 леммы 16.12 легко определить, является ли независимым заданное множество заданий (см. упр. 16.5.2).

Задача минимизации суммы штрафов за просроченные задания — это то же самое, что задача по максимизации суммы штрафов, которых удалось избежать бла-

годаря своевременному выполнению заданий. Таким образом, приведенная ниже теорема гарантирует, что с помощью жадного алгоритма можно найти независимое множество заданий A с максимальной суммой штрафов.

Теорема 16.13

Если S — множество единичных заданий с конечным сроком выполнения, а $\mathcal{I}$ — семейство всех независимых множеств заданий, то соответствующая система $(S, \mathcal{I})$ является матроидом.

Доказательство. Каждое подмножество независимого множества заданий также независимо. Чтобы доказать, что выполняется свойство обмена, предположим, что B и A — независимые множества заданий, и что $|B| > |A|$. Пусть k — наибольшее t, такое, что $N_t(B) \leq N_t(A)$ (такое значение t существует, поскольку $N_0(A) = N_0(B) = 0$). Поскольку $N_n(B) = |B|$ и $N_n(A) = |A|$, но $|B| > |A|$, получается, что $k < n$, и для всех j в диапазоне $k + 1 \leq j \leq n$ должно выполняться соотношение $N_j(B) > N_j(A)$. Следовательно, в множестве B содержится больше заданий с конечным сроком выполнения $k + 1$, чем в множестве A. Пусть a_i — задание из множества $B - A$ с конечным сроком выполнения $k + 1$ и пусть $A' = A \cup \{a_i\}$.

Теперь с помощью второго свойства леммы 16.12 покажем, что множество A' должно быть независимым. Поскольку множество A независимое, для любого $0 \leq t \leq k$ выполняется соотношение $N_t(A') = N_t(A) \leq t$. Для $k < t \leq n$, поскольку B — независимое множество, имеем $N_t(A') \leq N_t(B) \leq t$. Следовательно, множество A' независимое, что и завершает доказательство того, что $(S, \mathcal{I})$ — матроид. ■

Согласно теореме 16.11 для поиска независимого множества заданий A с максимальным весом можно использовать жадный алгоритм. После этого можно будет создать оптимальное расписание, в котором элементы множества A будут играть роль своевременных заданий. Этот метод дает эффективный алгоритм планирования единичных заданий с конечным сроком выполнения и штрафом для одного процессора. Время работы этого алгоритма, в котором используется процедура GREEDY, равно $O(n^2)$, поскольку каждая из $O(n)$ проверок независимости, выполняющихся в этом алгоритме, требует времени $O(n)$ (см. упр. 16.5.2). Более быструю реализацию этого алгоритма предлагается разработать в задаче 16.4.

На рис. 16.7 приведен пример задачи по планированию единичных заданий с конечным сроком выполнения и штрафом для одного процессора. В этом при-

	Задание						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Рис. 16.7. Пример задачи по планированию единичных заданий с конечным сроком выполнения и штрафом на одном процессоре.

16 Зак. 3726

мере жадный алгоритм поочередно выбирает задания a_1, a_2, a_3 и a_4, затем отвергает задания a_5 (поскольку $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) и a_6 (поскольку $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$) и наконец выбирает задание a_7. В результате получается оптимальное расписание

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle ,$$

которому соответствует общая сумма штрафа, равная $w_5 + w_6 = 50$.

Упражнения

16.5.1

Решите задачу планирования, параметры которой приведены на рис. 16.7, но каждый штраф w_i в которой заменен величиной $80 - w_i$.

16.5.2

Покажите, как с помощью второго свойства леммы 16.12 в течение времени $O(|A|)$ определить, независимо ли заданное множество заданий A.

Задачи

16.1. Размен монет

Рассмотрим задачу по выдаче сдачи, сумма которой составляет n копеек, использовав как можно меньше монет. Предполагается, что номинал каждой монеты выражается целым числом.

а. Опишите жадный алгоритм, в котором сдача выдается монетами номиналом 25 копеек, 10 копеек, 5 копеек и 1 копейка. Докажите, что алгоритм позволяет получить оптимальное решение.

б. Предположим, что номинал монет выражается степенями некоторого целого числа $c > 1$, т.е. номиналы представляют собой числа $c^0, c^1, \ldots, c^k$, где $k \geq 1$. Покажите, что жадный алгоритм всегда дает оптимальное решение.

в. Приведите пример множества номиналов монет, для которого жадный алгоритм не выдает оптимального решения. В это множество должна входить монета номиналом 1 копейка, чтобы решение существовало для любого значения n.

г. Разработайте алгоритм, позволяющий выдать сдачу с помощью любого набора монет, множество номиналов в котором состоит из k различных номиналов. Предполагается, что один из номиналов обязательно равен 1 копейке. Время работы алгоритма должно быть равно $O(nk)$.

16.2. Расписание, минимизирующее среднее время выполнения

Предположим, имеется множество заданий $S = \{a_1, a_2, \ldots, a_n\}$, в котором для выполнения задания a_i требуется p_i единиц времени. Имеется один компьютер, на котором будут выполняться эти задания и который способен одновременно выполнять не более одного задания. Пусть c_i — ***время завершения*** задания a_i, т.е. момент времени, когда прекращается его выполнение. Задача заключается в том, чтобы минимизировать среднее время завершения, т.е. величину $(1/n)\sum_{i=1}^{n} c_i$. Например, предположим, что имеется два задания, a_1 и a_2, которым соответствуют два значения времени выполнения: $p_1 = 3$ и $p_2 = 5$. Рассмотрим расписание, в котором сначала выполняется задание a_2, а затем — задание a_1. Тогда $c_2 = 5$, $c_1 = 8$ и среднее время завершения равно $(5+8)/2 = 6.5$. Если же первым выполняется задание a_1, то $c_1 = 3$, $c_2 = 8$, а среднее время завершения равно $(3+8)/2 = 5.5$.

а. Сформулируйте алгоритм, который планирует задания таким образом, чтобы минимизировать среднее время завершения. Каждое задание должно выполняться без прерываний, т.е., будучи запущенным, задание a_i должно непрерывно выполняться в течение p_i единиц времени. Докажите, что ваш алгоритм минимизирует среднее время завершения, и определите время его работы.

б. Предположим, что не все задания доступны сразу. Другими словами, с каждым заданием связано ***время выпуска*** (release time) r_i, раньше которого оно недоступно для обработки. Предположим также, что допускаются ***прерывания***, т.е. что задание может быть приостановлено и возобновлено позже. Например, задание a_i, время обработки которого равно $p_i = 6$, может быть запущено в момент времени 1 и приостановлено в момент времени 4. Затем оно может быть возобновлено в момент времени 10, еще раз приостановлено в момент времени 11, снова возобновлено в момент времени 13 и наконец завершено в момент времени 15. Таким образом, задание a_i в общей сложности выполняется в течение шести единиц времени, но время его выполнения разделено на три фрагмента. Время завершения задания a_i равно при таком расписании 15. Сформулируйте алгоритм, планирующий задания таким образом, чтобы минимизировать среднее время завершения в этом новом сценарии. Докажите, что ваш алгоритм минимизирует среднее время завершения, и найдите время его работы.

16.3. Ациклические подграфы

а. Для неориентированного графа $G = (V, E)$ ***матрицей инцидентности*** (incidence matrix) называется матрица M размером $|V| \times |E|$, такая, что $M_{ve} = 1$, если ребро e инцидентно вершине v, и $M_{ve} = 0$ в противном случае. Докажите, что множество столбцов матрицы M линейно независимо над полем целых чисел по модулю 2 тогда и только тогда, когда соответствующее множество ребер ациклично.

б. Предположим, что с каждым ребром неориентированного графа $G = (V, E)$ связан неотрицательный вес $w(e)$. Разработайте эффективный алгоритм поиска ациклического подмножества множества E с максимальным общим весом.

в. Пусть $G = (V, E)$ — произвольный ориентированный граф и пусть $(E, \mathcal{I})$ определено так, что $A \in \mathcal{I}$ тогда и только тогда, когда A не содержит ориентированных циклов. Приведите пример ориентированного графа G, такого, что связанная с ним система $(E, \mathcal{I})$ не является матроидом. Укажите, какое условие из определения матроида не соблюдается.

г. Для ориентированного графа $G = (V, E)$ ***матрицей инцидентности*** (incidence matrix) называется матрица M размером $|V| \times |E|$, такая, что $M_{ve} = -1$, если ребро e исходит из вершины v, $M_{ve} = +1$, если ребро e входит в вершину v, иначе $M_{ve} = 0$. Докажите, что если множество столбцов матрицы M линейно независимо, то соответствующее множество ребер не содержит ориентированных циклов.

д. В упр. 16.4.2 утверждается, что множество линейно независимых множеств произвольной матрицы M образует матроид. Объясните, почему результаты пп. (в) и (д) задачи не противоречат одно другому. В чем могут быть различия между понятием ациклического множества ребер и понятием множества связанных с ними линейно независимых столбцов матрицы инцидентности?

16.4. Вариации задачи планирования

Рассмотрим следующий алгоритм для решения задачи из раздела 16.5, в которой предлагается составить расписание единичных задач с конечными сроками выполнения и штрафами. Пусть все n отрезков времени изначально являются пустыми, где i-й интервал времени — это единичный промежуток времени, который заканчивается в момент i. Задачи рассматриваются в порядке монотонного убывания штрафов. Если при рассмотрении задания a_j существуют незаполненные отрезки времени, расположенные на шкале времени не позже конечного момента выполнения d_j этого задания, то задание a_j планируется для выполнения в течение последнего из этих отрезков, заполняя данный отрезок. Если же ни одного такого отрезка времени не осталось, задание a_j планируется для выполнения в течение последнего из незаполненных отрезков.

а. Докажите, что этот алгоритм всегда дает оптимальный ответ.

б. Эффективно реализуйте этот алгоритм с помощью леса непересекающихся множеств, описанного в разделе 21.3. Предполагается, что множество входных заданий уже отсортировано в порядке монотонного убывания штрафов. Проанализируйте время работы алгоритма в вашей реализации.

16.5. Кеширование

Современные компьютеры используют кеш для хранения в быстрой памяти данных небольших размеров. Несмотря на то что программа может работать

с большим количеством данных, хранение небольшого подмножества основной памяти в ***кеше*** — маленькой, но быстрой памяти — может существенно снизить общее время обращения к данным. При работе компьютерной программы она выполняет последовательность $\langle r_1, r_2, \ldots, r_n \rangle$ из n обращений к памяти, где каждое обращение запрашивает некоторый конкретный элемент данных. Например, программа, обращающаяся к четырем различным элементам $\{a, b, c, d\}$ может выполнять последовательность запросов $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Обозначим размер кеша как k. Когда кеш содержит k элементов и программа обращается к $(k+1)$-му элементу, система должна решить (для данного и каждого из последующих запросов), какие k элементов должны храниться в кеше. Точнее говоря, для каждого запроса r_i алгоритм управления кешированием проверяет, находится ли элемент r_i в кеше. Если да, то мы сталкиваемся с ***попаданием*** (cache hit); в противном случае мы имеем ***промах*** (cache miss). При промахе система запрашивает r_i из основной памяти, и алгоритм управления кешированием должен решить, следует ли хранить r_i в кеше. Если он принимает решение о хранении r_i, а кеш уже хранит k элементов, то из кеша следует убрать один элемент, чтобы освободить память для r_i. Алгоритм управления кешированием убирает данные из кеша таким образом, чтобы минимизировать количество промахов во всей последовательности запросов.

Обычно кеширование представляет собой оперативную (on-line) задачу, когда мы должны принимать решения о том, какие данные должны храниться в кеше, ничего не зная о будущих запросах. В данной же задаче мы рассматриваем автономную (оффлайн, off-line) версию этой задачи, в которой нам заранее известна вся последовательность n запросов и размер кеша k, и хотим минимизировать общее количество промахов кеша.

Эту задачу можно решить с применением жадной стратегии, которая выбирает для удаления элементы кеша, очередные обращения к которым в последовательности запросов будут выполняться позже всего.

а. Напишите псевдокод программы управления кешированием, использующей указанную стратегию. Входными данными должны быть последовательность запросов $\langle r_1, r_2, \ldots, r_n \rangle$ и размер кеша k, а на выходе должна получаться последовательность решений об удалениях из кеша (если таковые требуются) после каждого запроса. Чему равно время работы вашего алгоритма?

б. Покажите, что поставленная задача демонстрирует оптимальную подструктуру.

в. Докажите, что описанная выше стратегия обеспечивает минимально возможное количество промахов.

Заключительные замечания

Намного больше материала по жадным алгоритмам и матроидам можно найти в книгах Лоулера (Lawler) [223], а также Пападимитриу (Papadimitriou) и Штейглица (Steiglitz) [269].

Впервые жадный алгоритм был описан в литературе по комбинаторной оптимизации в статье Эдмондса (Edmonds) [100], опубликованной в 1971 году. Появление же теории матроидов датируется 1935 годом, когда вышла статья Уитни (Whitney) [353].

Приведенное здесь доказательство корректности жадного алгоритма для решения задачи о выборе процессов основано на доказательстве, предложенном Гаврилом (Gavril) [130]. Задача о планировании заданий изучалась Лоулером [223], Горовитцем (Horowitz), Сахни (Sahni) и Раясекараном (Rajasekaran) [180], а также Брассардом (Brassard) и Брейтли (Bratley) [53].

Коды Хаффмана были разработаны в 1952 году [184]. В работе Лелевера (Lelewer) и Хиршберга (Hirschberg) [230] имеется обзор методов сжатия данных, известных к 1987 году.

Развитие теории матроидов до теории гридоидов (greedoid) впервые было предложено Кортом (Korte) и Ловасом (Lovasz) [215–218], которые значительно обобщили представленную здесь теорию.

Глава 17. Амортизационный анализ

В ходе ***амортизационного анализа*** (amortized analysis) время, необходимое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям. Этот анализ можно использовать, например, чтобы показать, что, даже если одна из операций последовательности является дорогостоящей, при усреднении по всей последовательности средняя стоимость операций будет небольшой. Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность. При выполнении амортизационного анализа гарантируется *средняя производительность операций в наихудшем случае.*

В первых трех разделах этой главы описываются три наиболее распространенных метода, которые используются при амортизационном анализе. Начало раздела 17.1 посвящено групповому анализу, в ходе которого определяется верхняя граница $T(n)$ полной стоимости последовательности n операций. Тогда средняя стоимость операции вычисляется как $T(n)/n$. Мы принимаем среднюю стоимость равной амортизированной стоимости каждой операции, так что амортизированная стоимость всех операций одинакова.

В разделе 17.2 описывается метод бухгалтерского учета, в котором определяется амортизированная стоимость каждой операции. Если имеется несколько видов операций, то каждый из них может характеризоваться своей амортизированной стоимостью. В этом методе стоимость некоторых операций, находящихся в начальной части последовательности, может переоцениваться. Эта переоценка рассматривается как своего рода "предварительный кредит" для определенных объектов структуры данных. Впоследствии такой кредит используется для "оплаты" тех операций последовательности, которые оцениваются ниже, чем стоят на самом деле.

В разделе 17.3 описан метод потенциалов, похожий на метод бухгалтерского учета в том отношении, что в нем определяется амортизированная стоимость каждой операции, причем стоимость ранних операций последовательности может переоцениваться, чтобы впоследствии компенсировать заниженную стоимость более поздних операций. В методе потенциалов кредит поддерживается как "потенциальная энергия" структуры данных в целом, а не связывается с отдельными объектами этой структуры данных.

Все три перечисленных метода будут опробованы на двух примерах. В одном из них рассматривается стек с дополнительной операцией MULTIPOP, в ходе ко-

торой со стека одновременно снимается несколько объектов. Во втором примере реализуется бинарный счетчик, ведущий счет от нуля с помощью единственной операции — INCREMENT.

Изучая эту главу, следует иметь в виду, что при выполнении амортизационного анализа расходы начисляются только для анализа алгоритма и что в коде в них нет никакой необходимости. Например, если при использовании метода бухгалтерского учета объекту x приписывается какой-нибудь кредит, то в коде не нужно присваивать соответствующую величину некоторому атрибуту наподобие $x.credit$.

Глубокое понимание той или иной структуры данных, возникающей в ходе амортизационного анализа, может помочь оптимизировать ее устройство. Например, в разделе 17.4 с помощью метода потенциалов проводится анализ динамически увеличивающейся и уменьшающейся таблицы.

17.1. Групповой анализ

В ходе ***группового анализа*** (aggregate analysis) исследователь показывает, что в *наихудшем случае* суммарное время выполнения последовательности всех n операций равно $T(n)$. Поэтому в наихудшем случае средняя, или ***амортизированная, стоимость*** (amortized cost), приходящаяся на одну операцию, определяется соотношением $T(n)/n$. Заметим, что такая амортизированная стоимость применима ко всем операциям, даже если в последовательности имеется несколько разных их типов. В других двух методах, которые изучаются в этой главе (методе бухгалтерского учета и методе потенциалов), операциям различного вида могут присваиваться разные амортизированные стоимости.

Стековые операции

В качестве первого примера группового анализа рассматриваются стеки, в которых реализована дополнительная операция. В разделе 10.1 представлены две основные стековые операции, для выполнения каждой из которых требуется время $O(1)$.

PUSH(S, x) добавляет объект x в стек S.

POP(S) снимает объект с вершины стека S и возвращает его. Вызов POP с пустым стеком генерирует ошибку.

Поскольку каждая из этих операций выполняется в течение времени $O(1)$, примем их длительность за единицу. Тогда полная стоимость последовательности, состоящей из n операций PUSH и POP, равна n, а фактическое время выполнения n таких операций — $\Theta(n)$.

Теперь добавим к стеку операцию MULTIPOP(S, k), снимающую k объектов (или все оставшиеся, если всего их меньше, чем k) с вершины стека S . В приведенном ниже псевдокоде операция STACK-EMPTY возвращает значение TRUE,

(а)	(б)	(в)
top → 23		
17		
6		
39		
10	top → 10	
47	47	
—	—	—

Рис. 17.1. Работа процедуры MULTIPOP со стеком S, начальное состояние которого показано в части **(а)**. Процедура MULTIPOP($S, 4$) снимает со стека верхние четыре объекта, после чего стек принимает вид, показанный в части **(б)**. Очередная операция, MULTIPOP($S, 7$), опустошает стек, как показано в части **(в)**, поскольку перед ней в стеке оставалось менее семи объектов.

если в данный момент в стеке нет объектов, и значение FALSE — в противном случае.

```
MULTIPOP(S, k)
1  while STACK-EMPTY(S) == FALSE и k > 0
2      POP(S)
3      k = k - 1
```

На рис. 17.1 показан пример работы процедуры MULTIPOP.

В течение какого времени операция MULTIPOP(S, k) выполняется над стеком, содержащим s объектов? Фактическое время работы линейно зависит от количества реально выполняемых операций POP, поэтому достаточно проанализировать процедуру MULTIPOP в терминах абстрактных единичных стоимостей каждой из операций PUSH и POP. Количество итераций цикла **while** равно количеству $\min(s, k)$ объектов, снимаемых со стека. При каждой итерации в строке 2 выполняется однократный вызов процедуры POP. Таким образом, полная стоимость процедуры MULTIPOP равна $\min(s, k)$, а фактическое время работы является линейной функцией от этой величины.

Теперь проанализируем последовательность операций PUSH, POP и MULTIPOP, действующих на изначально пустой стек. Стоимость операции MULTIPOP в наихудшем случае равна $O(n)$, поскольку в стеке не более n объектов. Таким образом, время работы любой стековой операции в наихудшем случае равно $O(n)$, поэтому стоимость последовательности n операций равна $O(n^2)$, так как эта последовательность может содержать $O(n)$ операций MULTIPOP, стоимость каждой из которых равна $O(n)$. Но несмотря на то что анализ проведен правильно, результат $O(n^2)$, полученный при рассмотрении наихудшей стоимости каждой операции в отдельности, является слишком грубым.

С помощью группового анализа можно получить более точную верхнюю границу при рассмотрении совокупной последовательности n операций. Фактически, хотя одна операция MULTIPOP может быть весьма дорогостоящей, стоимость произвольной последовательности, состоящей из n операций PUSH, POP и MULTIPOP, которая выполняется над изначально пустым стеком, не превышает $O(n)$. Почему? Потому что каждый помещенный в стек объект можно извлечь оттуда не более одного раза. Таким образом, число вызовов операций POP (вклю-

чая их вызовы в процедуре MULTIPOP) для непустого стека не может превышать количество выполненных операций PUSH, которое, в свою очередь, не больше n. При любом n для выполнения произвольной последовательности из n операций PUSH, POP и MULTIPOP требуется суммарное время $O(n)$. Таким образом, средняя стоимость операции равна $O(n)/n = O(1)$. В групповом анализе амортизированная стоимость каждой операции принимается равной ее средней стоимости, поэтому в данном примере все три стековые операции характеризуются амортизированной стоимостью, равной $O(1)$.

Еще раз стоит заметить, что хотя только что было показано, что средняя стоимость (а следовательно, и время выполнения) стековой операции равна $O(1)$, при этом не делалось никаких вероятностных рассуждений. Фактически была найдена граница времени выполнения последовательности из n операций *в наихудшем случае*, равная $O(n)$. После деления этой полной стоимости на n получается средняя стоимость одной операции, или амортизированная стоимость.

Увеличение показаний бинарного счетчика

В качестве другого примера группового анализа рассмотрим задачу реализации k-битового бинарного счетчика, который ведет счет от нуля в восходящем направлении. В качестве счетчика используется битовый массив $A[0\,..\,k-1]$, где $A.length = k$. Младший бит хранящегося в счетчике бинарного числа x находится в элементе $A[0]$, а старший бит — в элементе $A[k-1]$, так что $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Изначально $x = 0$, т.е. $A[i] = 0$ для $i = 0, 1, \ldots, k-1$. Чтобы увеличить показания счетчика на 1 (по модулю 2^k), используется следующая процедура.

```
INCREMENT(A)
1  i = 0
2  while i < A.length и A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

На рис. 17.2 показано, что происходит в бинарном счетчике при его увеличении 16 раз. В начале каждой итерации цикла **while** в строках 2–4 мы добавляем 1 к биту в позиции i. Если $A[i] = 1$, то добавление 1 обнуляет бит, который находится на позиции i, и приводит к тому, что добавление 1 будет выполнено и в позиции $i + 1$ на следующей операции цикла. В противном случае цикл оканчивается, так что если по его окончании $i < k$, то $A[i] = 0$ и нам нужно изменить значение i-го бита на 1, что и делается в строке 6. Стоимость каждой операции INCREMENT линейно зависит от количества измененных битов.

Как и в примере со стеком, поверхностный анализ даст правильную, но неточную оценку. В наихудшем случае, когда массив A состоит только из единиц, для выполнения операции INCREMENT потребуется время $\Theta(k)$. Таким образом, выполнение последовательности из n операций INCREMENT для изначально обнуленного счетчика в наихудшем случае займет время $O(nk)$.

Значение счетчика	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Общая стоимость
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Рис. 17.2. 8-битовый бинарный счетчик, значение которого увеличивается с 0 до 16 с помощью 16 операций INCREMENT. Биты, значения которых изменяются при переходе к следующему значению, заштрихованы. Справа приведена стоимость, требуемая для изменения битов. Обратите внимание, что полная стоимость никогда более чем в два раза не превышает общего количества операций INCREMENT.

Этот анализ можно уточнить, в результате чего для последовательности из n операций INCREMENT в наихудшем случае получается стоимость $O(n)$. Такая оценка возможна благодаря тому, что далеко не при каждом вызове процедуры INCREMENT изменяются значения всех битов. Как видно из рис. 17.2, элемент $A[0]$ изменяется при каждом вызове операции INCREMENT. Следующий по старшинству бит $A[1]$ изменяется только через раз, так что последовательность из n операций INCREMENT над изначально обнуленным счетчиком приводит к изменению элемента $A[1]$ $\lfloor n/2 \rfloor$ раз. Аналогично бит $A[2]$ изменяется только каждый четвертый раз, т.е. $\lfloor n/4 \rfloor$ раз в последовательности из n операций INCREMENT над изначально обнуленным счетчиком. В общем случае для $i = 0, 1, \ldots, k-1$ бит $A[i]$ изменяется $\lfloor n/2^i \rfloor$ раз в последовательности из n операций INCREMENT над изначально обнуленным счетчиком. Биты же в позициях $i \geq k$ не изменяются. Таким образом, общее количество изменений битов при выполнении последовательности операций равно

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

согласно уравнению (А.6). Поэтому время выполнения последовательности из n операций INCREMENT над изначально обнуленным счетчиком в наихудшем случае равно $O(n)$. Средняя стоимость каждой операции, а следовательно, и амортизированная стоимость операции, равна $O(n)/n = O(1)$.

Упражнения

17.1.1

Остается ли справедливой оценка амортизированной стоимости стековых операций, равная $O(1)$, если включить в множество стековых операций операцию MULTIPUSH, помещающую в стек k элементов?

17.1.2

Покажите, что если бы в пример с k-битовым счетчиком была включена операция DECREMENT, стоимость n операций была бы равной $\Theta(nk)$.

17.1.3

Предположим, что над структурой данных выполняется n операций. Стоимость i-й по порядку операции равна i, если i — это точная степень двойки, и 1 — в противном случае. Определите с помощью группового анализа амортизированную стоимость операции.

17.2. Метод бухгалтерского учета

В ***методе бухгалтерского учета*** (accounting method), применяемом в ходе группового анализа, разные операции оцениваются по-разному, в зависимости от их фактической стоимости. Величина, которая начисляется на операцию, называется ***амортизированной стоимостью*** (amortized cost). Если амортизированная стоимость операции превышает ее фактическую стоимость, то соответствующая разность присваивается определенным объектам структуры данных как ***кредит*** (credit). Кредит можно использовать впоследствии для компенсирующих выплат на операции, амортизированная стоимость которых меньше их фактической стоимости. Таким образом, можно полагать, что амортизированная стоимость операции состоит из ее фактической стоимости и кредита, который либо накапливается, либо расходуется. Этот метод существенно отличается от группового анализа, в котором все операции характеризуются одинаковой амортизированной стоимостью.

К выбору амортизированной стоимости следует подходить с осторожностью. Если нужно провести анализ с использованием амортизированной стоимости, чтобы показать, что в наихудшем случае средняя стоимость операции невелика, полная амортизированная стоимость последовательности операций должна быть верхней границей полной фактической стоимости последовательности. Более того, как и в групповом анализе, это соотношение должно соблюдаться для всех последовательностей операций. Если обозначить фактическую стоимость i-й операции через c_i, а амортизированную стоимость i-й операции через $\widehat{c}_i$, то указанное требование для всех последовательностей, состоящих из n операций, можно

выразить следующим образом:

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \ . \tag{17.1}$$

Общий кредит, хранящийся в структуре данных, представляет собой разность между полной амортизированной стоимостью и полной фактической стоимостью, или $\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i$. Согласно неравенству (17.1) полный кредит, связанный со структурой данных, все время должен быть неотрицательным. Если бы полный кредит в каком-либо случае мог стать отрицательным (в результате недооценки ранних операций с надеждой восполнить счет впоследствии), то полная амортизированная стоимость в тот момент была бы ниже соответствующей фактической стоимости; значит, для последовательности операций полная амортизированная стоимость не была бы в этот момент времени верхней границей полной фактической стоимости. Таким образом, необходимо позаботиться о том, чтобы полный кредит для структуры данных никогда не становился отрицательным.

Стековые операции

Чтобы проиллюстрировать, как проводится амортизационный анализ методом бухгалтерского учета, вернемся к примеру со стеком. Напомним, что фактическая стоимость операций была следующей:

PUSH 1 ,
POP 1 ,
MULTIPOP $\min(k, s)$,

где k — аргумент процедуры MULTIPOP, а s — размер стека в момент ее вызова. Присвоим приведенные ниже амортизированные стоимости:

PUSH 2 ,
POP 0 ,
MULTIPOP 0 .

Заметим, что амортизированная стоимость операции MULTIPOP равна константе (0), в то время как ее фактическая стоимость — переменная величина. В этой схеме все три амортизированные стоимости равны $O(1)$, хотя в общем случае асимптотическое поведение амортизированных стоимостей рассматриваемых операций может быть разным.

Теперь покажем, что любую последовательность стековых операций можно оплатить путем начисления амортизированных стоимостей. Предположим, что для представления каждой единицы затрат используется денежный счет. Пусть изначально стек пуст. Вспомним аналогию между стеком как структурой данных и стопкой тарелок в кафетерии, приведенную в разделе 10.1. При добавлении тарелки в стопку 1 доллар затрачивается на оплату самой операции добавления, а еще 1 доллар (из двух начисленных на операцию PUSH) остается в запасе, как бы на этой тарелке. В любой момент времени каждой тарелке стека соответствует 1 доллар кредита.

Хранящийся в тарелке 1 доллар предназначен для оплаты стоимости ее извлечения из стека. На операцию POP ничего не начисляется, а ее фактическая

стоимость выплачивается за счет кредита, который хранится в стеке. При извлечении тарелки изымается 1 доллар кредита, который используется для оплаты фактической стоимости операции. Таким образом, благодаря небольшой переоценке операции PUSH отпадает необходимость начислять какую-либо сумму на операцию POP.

Более того, на операцию MULTIPOP также не нужно начислять никакой суммы. При извлечении первой тарелки мы берем из нее 1 доллар кредита и тратим его на оплату фактической стоимости операции POP. При извлечении второй тарелки в ней также есть 1 доллар кредита на оплату фактической стоимости операции POP и т.д. Таким образом, начисленной заранее суммы всегда достаточно для оплаты операций MULTIPOP. Другими словами, поскольку с каждой тарелкой стека связан 1 доллар кредита и в стеке всегда содержится неотрицательное количество тарелок, можно быть уверенным, что сумма кредита всегда неотрицательна. Таким образом, для любой последовательности из n операций PUSH, POP и MULTIPOP полная амортизированная стоимость является верхней границей полной фактической стоимости. Поскольку полная амортизированная стоимость равна $O(n)$, полная фактическая стоимость также определяется этим значением.

Увеличение показаний бинарного счетчика

В качестве другого примера, иллюстрирующего метод бухгалтерского учета, проанализируем операцию INCREMENT, которая выполняется над бинарным изначально обнуленным счетчиком. Ранее мы убедились, что время выполнения этой операции пропорционально количеству битов, изменяющих свое значение. В данном примере это количество используется в качестве стоимости. Как и в предыдущем примере, для представления каждой единицы затрат (в данном случае — изменения битов) будет использован денежный счет.

Чтобы провести амортизационный анализ, начислим на операцию, при которой биту присваивается значение 1 (т.е. бит устанавливается), амортизированную стоимость, равную 2 долларам. Когда бит устанавливается, 1 доллар (из двух начисленных) расходуется на оплату операции по самой установке. Оставшийся 1 доллар вкладывается в этот бит в качестве кредита для последующего использования при его обнулении. В любой момент времени с каждой единицей содержащегося в счетчике значения связан 1 доллар кредита, поэтому для обнуления бита нет необходимости начислять какую-либо сумму; за сброс бита достаточно будет уплатить 1 доллар.

Теперь можно определить амортизированную стоимость операции INCREMENT. Стоимость обнуления битов в цикле **while** выплачивается за счет тех денег, которые связаны с этими битами. В процедуре INCREMENT устанавливается не более одного бита (в строке 6), поэтому амортизированная стоимость операции INCREMENT не превышает 2 долларов. Количество единиц в бинарном числе, представляющем показания счетчика, не может быть отрицательным, поэтому и сумма кредита всегда неотрицательна. Таким образом,

полная амортизированная стоимость n операций INCREMENT равна $O(n)$. Это и есть оценка полной фактической стоимости.

Упражнения

17.2.1

Предположим, что над стеком выполняется последовательность операций; размер стека при этом никогда не превышает k. После каждых k операций проводится резервное копирование стека. Присвоив различным стековым операциям соответствующие амортизированные стоимости, покажите, что стоимость n стековых операций, включая копирование стека, равна $O(n)$.

17.2.2

Выполните упр. 17.1.3, применив для анализа метод бухгалтерского учета.

17.2.3

Предположим, что нам нужно иметь возможность не только увеличивать показания счетчика, но и сбрасывать его (т.е. делать так, чтобы значения всех битов были равны 0). Считая, что время проверки или модификации одного бита составляет $\Theta(1)$, покажите, как осуществить реализацию счетчика в виде массива битов, чтобы для выполнения произвольной последовательности из n операций INCREMENT и RESET над изначально обнуленным счетчиком потребовалось бы время $O(n)$. (*Указание:* отслеживайте в счетчике самый старший разряд, содержащий единицу.)

17.3. Метод потенциалов

Вместо представления предоплаченной работы в виде кредита, хранящегося в структуре данных вместе с отдельными объектами, в ходе амортизированного анализа по ***методу потенциалов*** (potential method) такая работа представляется в виде "потенциальной энергии", или просто "потенциала", который можно высвободить для оплаты последующих операций. Этот потенциал связан со структурой данных в целом, а не с ее отдельными объектами.

Метод потенциалов работает следующим образом. Мы начинаем с исходной структуры данных D_0, над которой выполняется n операций. Для всех $i = 1, 2, \ldots, n$ обозначим через c_i фактическую стоимость i-й операции, а через D_i — структуру данных, которая получается в результате применения i-й операции к структуре данных D_{i-1}. ***Функция потенциала*** (potential function) Φ отображает каждую структуру данных D_i на действительное число $\Phi(D_i)$, которое является ***потенциалом*** (potential), связанным со структурой данных D_i. ***Амортизированная стоимость*** (amortized cost) $\widehat{c}_i$ i-й операции определяется соотношением

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \qquad (17.2)$$

Таким образом, амортизированная стоимость каждой операции представляет собой ее фактическую стоимость плюс приращение потенциала в результате выполнения операции. Согласно уравнению (17.2) полная амортизированная стоимость n операций равна

$$\begin{aligned}\sum_{i=1}^{n}\widehat{c}_i &= \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\ . \end{aligned} \quad (17.3)$$

Второе равенство следует из уравнения (A.9), поскольку слагаемые $\Phi(D_i)$ являются телескопическими.

Если функцию потенциала Φ можно определить таким образом, чтобы выполнялось неравенство $\Phi(D_n) \geq \Phi(D_0)$, то полная амортизированная стоимость $\sum_{i=1}^{n}\widehat{c}_i$ является верхней границей полной фактической стоимости $\sum_{i=1}^{n} c_i$. На практике не всегда известно, сколько операций может быть выполнено, поэтому, если наложить условие $\Phi(D_i) \geq \Phi(D_0)$ для всех i, то, как и в методе бухгалтерского учета, будет обеспечена предоплата. Часто удобно определить величину $\Phi(D_0)$ равной нулю, а затем показать, что для всех i выполняется неравенство $\Phi(D_i) \geq 0$. (См. упр. 17.3.1, в котором идет речь о том, как быть в ситуации, когда $\Phi(D_0) \neq 0$.)

Интуитивно понятно, что если разность потенциалов $\Phi(D_i) - \Phi(D_{i-1})$ для i-й операции положительна, то амортизированная стоимость $\widehat{c}_i$ представляет переоценку i-й операции и потенциал структуры данных возрастает. Если разность потенциалов отрицательна, то амортизированная стоимость представляет недооценку i-й операции и фактическая стоимость операции выплачивается за счет уменьшения потенциала.

Амортизированные стоимости, определенные уравнениями (17.2) и (17.3), зависят от выбора функции потенциала Φ. Амортизированные стоимости, соответствующие разным функциям потенциала, могут различаться, при этом все равно оставаясь верхними границами фактических стоимостей. При выборе функций потенциалов часто допустимы компромиссы; то, какую функцию лучше всего использовать, зависит от оценки времени, которую нужно получить.

Стековые операции

Чтобы проиллюстрировать метод потенциалов, еще раз обратимся к примеру со стековыми операциями PUSH, POP и MULTIPOP. Определим функцию потенциала Φ для стека как количество объектов в этом стеке. Для пустого стека D_0, с которого мы начинаем, выполняется соотношение $\Phi(D_0) = 0$. Поскольку количество объектов в стеке не может быть отрицательным, стеку D_i, полученному в результате выполнения i-й операции, соответствует неотрицательный

потенциал; следовательно,

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0) \ .\end{aligned}$$

Таким образом, полная амортизированная стоимость n операций, связанная с функцией Φ, представляет собой верхнюю границу фактической стоимости.

Теперь вычислим амортизированные стоимости различных стековых операций. Если i-я операция над стеком, содержащим s объектов, — операция PUSH, то разность потенциалов равна

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \ .\end{aligned}$$

Согласно уравнению (17.2) амортизированная стоимость операции PUSH равна

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 \ .\end{aligned}$$

Предположим, что i-я операция над стеком — операция MULTIPOP(S, k), и что из него извлекается $k' = \min(k, s)$ объектов. Фактическая стоимость этой операции равна k', а разность потенциалов —

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' \ .$$

Таким образом, амортизированная стоимость операции MULTIPOP равна

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 \ .\end{aligned}$$

Аналогично можно показать, что амортизированная стоимость операции POP также равна 0.

Амортизированная стоимость каждой из трех операций равна $O(1)$, поэтому полная амортизированная стоимость последовательности из n операций равна $O(n)$. Мы уже доказали, что $\Phi(D_i) \geq \Phi(D_0)$, поэтому полная амортизированная стоимость n операций является верхней границей полной фактической стоимости. Поэтому в наихудшем случае стоимость n операций равна $O(n)$.

Увеличение показаний бинарного счетчика

В качестве другого примера применения метода потенциалов снова обратимся к задаче о приращении показаний бинарного счетчика. На этот раз определим потенциал счетчика после выполнения i-й операции INCREMENT как количество b_i содержащихся в счетчике единиц после этой операции.

Вычислим амортизированную стоимость операции INCREMENT. Предположим, что i-я операция INCREMENT обнуляет t_i бит. В таком случае фактическая стоимость этой операции не превышает значения $t_i + 1$, поскольку вдобавок к обнулению t_i бит значение 1 присваивается не более чем одному биту. Если $b_i = 0$, то в ходе выполнения i-й операции обнуляются все k бит, так что $b_{i-1} = t_i = k$. Если $b_i > 0$, то выполняется соотношение $b_i = b_{i-1} - t_i + 1$. В любом случае справедливо неравенство $b_i \leq b_{i-1} - t_i + 1$ и разность потенциалов равна

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \ .\end{aligned}$$

Таким образом, амортизированная стоимость определяется соотношением

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 \ .\end{aligned}$$

Если вначале показания счетчика равны нулю, то $\Phi(D_0) = 0$. Поскольку при всех i справедливо неравенство $\Phi(D_i) \geq 0$, полная амортизированная стоимость последовательности из n операций INCREMENT является верхней границей полной фактической стоимости, поэтому в наихудшем случае стоимость n операций INCREMENT равна $O(n)$.

Метод потенциалов предоставляет простой способ анализа счетчика даже в том случае, когда отсчет начинается не с нуля. Пусть изначально показание счетчика содержит b_0 единиц, а после выполнения n операций INCREMENT — b_n единиц, где $0 \leq b_0, b_n \leq k$. (Напомним, что k — количество битов в счетчике.) Тогда уравнение (17.3) можно переписать следующим образом:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c}_i - \Phi(D_n) + \Phi(D_0) \ . \tag{17.4}$$

При всех $1 \leq i \leq n$ выполняется неравенство $\widehat{c}_i \leq 2$. В силу соотношений $\Phi(D_0) = b_0$ и $\Phi(D_n) = b_n$ полная фактическая стоимость n операций INCREMENT равна

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 \ .\end{aligned}$$

В частности, заметим, что, поскольку $b_0 \leq k$, при $k = O(n)$ полная фактическая стоимость равна $O(n)$. Другими словами, если выполняется не менее $n = \Omega(k)$ операций INCREMENT, полная фактическая стоимость независимо от начального показания счетчика равна $O(n)$.

Упражнения

17.3.1

Предположим, задана такая функция потенциала Φ, что при всех i выполняется неравенство $\Phi(D_i) \geq \Phi(D_0)$, но $\Phi(D_0) \neq 0$. Покажите, что существует функция потенциала Φ', такая, что $\Phi'(D_0) = 0$ и $\Phi'(D_i) \geq 0$ для всех $i \geq 1$, и соответствующая функции Φ' амортизированная стоимость такая же, как и для функции Φ.

17.3.2

Еще раз выполните упр. 17.1.3, используя метод потенциалов.

17.3.3

Рассмотрим обычную n-элементную бинарную неубывающую пирамиду. Пусть в этой структуре данных поддерживаются операции INSERT и EXTRACT-MIN, для выполнения которых в наихудшем случае требуется время $O(\lg n)$. Определите функцию потенциала Φ, такую, что амортизированная стоимость операции INSERT равна $O(\lg n)$, а амортизированная стоимость операции EXTRACT-MIN — $O(1)$, и покажите, что она работает.

17.3.4

Чему равна полная стоимость выполнения n стековых операций PUSH, POP и MULTIPOP, если предположить, что в начале стек содержит s_0 объектов, а в конце — s_n объектов?

17.3.5

Предположим, что изначально показание счетчика не равно нулю, а определяется числом, содержащим в двоичном представлении b единиц. Покажите, что стоимость выполнения n операций INCREMENT равна $O(n)$ при условии, что $n = \Omega(b)$. (Не следует предполагать, что b — константа.)

17.3.6

Покажите, как реализовать очередь на основе двух обычных стеков (см. упр. 10.1.6) таким образом, чтобы амортизированная стоимость каждой из операций ENQUEUE и DEQUEUE была равна $O(1)$.

17.3.7

Разработайте структуру данных для поддержки следующих двух операций над динамическим мультимножеством целых чисел S, в котором могут содержаться одинаковые значения:

INSERT(S, x), вставляющая x в S.

DELETE-LARGER-HALF(S), удаляющая наибольшие $\lceil |S| / 2 \rceil$ элементов из S.

Поясните, как реализовать эту структуру данных, чтобы любая последовательность из m указанных операций выполнялась за время $O(m)$. Ваша реализация должна также включать способ вывода элементов S за время $O(|S|)$.

17.4. Динамические таблицы

В некоторых приложениях заранее не известно, сколько элементов будет храниться в таблице. Может возникнуть ситуация, когда для таблицы выделяется место, а впоследствии оказывается, что его недостаточно. В этом случае приходится выделять больший объем памяти и копировать все объекты из исходной таблицы в новую, большего размера. Аналогично, если из таблицы удаляется много объектов, может понадобиться преобразовать ее в таблицу меньшего размера. В этом разделе исследуется задача о динамическом расширении и сжатии таблицы. Методом амортизационного анализа будет показано, что амортизированная стоимость вставки и удаления равна всего лишь $O(1)$, даже если фактическая стоимость операции больше из-за того, что она приводит к расширению или сжатию таблицы. Более того, мы выясним, как обеспечить соблюдение условия, при котором неиспользованное пространство динамической таблицы не превышает фиксированную долю ее полного пространства.

Предполагается, что в динамической таблице поддерживаются операции TABLE-INSERT и TABLE-DELETE. В результате выполнения операции TABLE-INSERT в таблицу добавляется элемент, занимающий одну ***ячейку*** (slot), — пространство для одного элемента. Аналогично операцию TABLE-DELETE можно представлять как удаление элемента из таблицы, в результате чего одна ячейка освобождается. Подробности метода, с помощью которого организуется таблица, не важны; можно воспользоваться стеком (раздел 10.1), пирамидой (глава 6) или хеш-таблицей (глава 11). Для реализации хранилища объектов можно также использовать массив или набор массивов, как это было сделано в разделе 10.3.

Оказывается, достаточно удобно применить концепцию, введенную при выполнении анализа хеширования (глава 11). Определим ***коэффициент заполнения*** (load factor) $\alpha(T)$ непустой таблицы T как количество хранящихся в них элементов, деленное на ее размер (измеряемый в количестве ячеек). Размер пустой таблицы (в которой нет ни одного элемента) примем равным нулю, а ее коэффициент заполнения — единице. Если коэффициент заполнения динамической таблицы ограничен снизу константой, ее неиспользованное пространство никогда не превышает фиксированной части ее полного размера.

Начнем анализ с динамической таблицы, в которую выполняются только вставки. Впоследствии будет рассмотрен более общий случай, когда разрешены и вставки, и удаления.

17.4.1. Расширение таблицы

Предположим, что место для хранения таблицы выделяется в виде массива ячеек. Таблица становится заполненной, когда заполняются все ее ячейки или (что

эквивалентно) когда ее коэффициент заполнения становится равным 1[1]. В некоторых программных средах при попытке вставить элемент в заполненную таблицу не остается ничего другого, как прибегнуть к аварийному завершению программы, сопровождаемому выдачей сообщения об ошибке. Однако мы предполагаем, что наша программная среда, подобно многим современным средам, обладает системой управления памятью, позволяющей по запросу выделять и освобождать блоки памяти. Таким образом, когда в заполненную таблицу вставляется элемент, ее можно ***расширить*** (expand), выделив место для новой таблицы, содержащей больше ячеек, чем было в старой. Поскольку таблица всегда должна размещаться в непрерывной области памяти, для большей таблицы необходимо выделить новый массив, а затем скопировать элементы из старой таблицы в новую.

Общепринятый эвристический подход заключается в том, чтобы в новой таблице было в два раза больше ячеек, чем в старой. Если в таблицу элементы только вставляются, то значение ее коэффициента заполнения будет не меньше $1/2$, поэтому объем неиспользованного места никогда не превысит половины полного размера таблицы.

В приведенном ниже псевдокоде предполагается, что T — объект, представляющий таблицу. Атрибут $T.table$ содержит указатель на блок памяти, представляющий таблицу. В $T.num$ содержится количество элементов в таблице, а в $T.size$ — полное количество ячеек в таблице. Изначально таблица пустая: $T.num = T.size = 0$.

TABLE-INSERT(T, x)

1 **if** $T.size == 0$
2   Выделить $T.table$ с 1 ячейкой
3     $T.size = 1$
4 **if** $T.num == T.size$
5   Выделить $new\text{-}table$ с $2 \cdot T.size$ ячейками
6   Вставить все элементы из $T.table$ в $new\text{-}table$
7   Освободить $T.table$
8     $T.table = new\text{-}table$
9     $T.size = 2 \cdot T.size$
10 Вставить x в $T.table$
11 $T.num = T.num + 1$

Обратите внимание, что здесь имеется две "вставки" (сама процедура TABLE-INSERT и операция ***элементарной вставки*** (elementary insertion) в таблицу), выполняемые в строках 6 и 10. Время работы процедуры TABLE-INSERT можно проанализировать в терминах количества элементарных вставок, считая стоимость каждой такой операции равной 1. Предполагается, что фактическое время работы процедуры TABLE-INSERT линейно зависит от времени вставки отдельных эле-

---

[1] В некоторых случаях, например при работе с хешированными таблицами с открытой адресацией, может понадобиться считать таблицу заполненной, если ее коэффициент заполнения равен некоторой константе, строго меньшей 1. (См. упр. 17.4.1.)

ментов, так что накладные расходы на выделение исходной таблицы в строке 2 — константа, а накладные расходы на выделение и освобождение памяти в строках 5 и 7 пренебрежимо малы по сравнению со стоимостью переноса элементов в строке 6. Назовем ***расширением*** (expansion) событие, при котором выполняются строки 5–9.

Проанализируем последовательность n операций TABLE-INSERT, которые выполняются над изначально пустой таблицей. Чему равна стоимость c_i i-й операции? Если в данный момент в таблице имеется свободное место (или если это первая операция), то $c_i = 1$, поскольку достаточно выполнить лишь одну элементарную операцию вставки в строке 10. Если же таблица заполнена и нужно выполнить ее расширение, то $c_i = i$: стоимость равна 1 для элементарной вставки в строке 10 плюс $i - 1$ для элементов, которые необходимо скопировать из старой таблицы в новую (строка 6). Если выполняется n операций, то стоимость последней из них в наихудшем случае равна $O(n)$, в результате чего верхняя граница полного времени выполнения n операций равна $O(n^2)$.

Эта оценка является неточной, поскольку при выполнении n операций TABLE-INSERT потребность в расширении таблицы возникает не так часто. Точнее говоря, i-я операция приводит к расширению, только если величина $i - 1$ равна степени 2. С помощью группового анализа можно показать, что амортизированная стоимость операции в действительности равна $O(1)$. Стоимость i-й операции равна

$$c_i = \begin{cases} i \ , & \text{если } i-1 \text{ является точной степенью } 2 \ , \\ 1 & \text{в противном случае} \ . \end{cases}$$

Таким образом, полная стоимость выполнения n операций TABLE-INSERT равна

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \ , \end{aligned}$$

поскольку стоимость не более n операций равна 1, а стоимости остальных операций образуют геометрическую прогрессию. Поскольку полная стоимость n операций TABLE-INSERT ограничена величиной $3n$, амортизированная стоимость одной операции не превышает 3.

С помощью метода бухгалтерского учета можно показать "на пальцах", что амортизированная стоимость операции TABLE-INSERT должна быть равна 3. Интуитивно понятно, что для каждого элемента приходится трижды платить за элементарную операцию вставки: за саму вставку в текущую таблицу, за перемещение этого элемента при расширении таблицы и за перемещение еще одного элемента, который уже однажды был перемещен в ходе расширения таблицы. Например, предположим, что сразу после расширения таблицы ее размер становится равным m. Тогда количество элементов в ней равно $m/2$ и таблице соответствует нулевой кредит. На каждую вставку начисляется по 3 доллара. Элементарная

вставка, которая выполняется тут же, стоит 1 доллар. Еще 1 доллар зачисляется в кредит на вставляемый элемент. Третий доллар зачисляется в качестве кредита на один из уже содержащихся в таблице $m/2$ элементов. Для заполнения таблицы требуется $m/2 - 1$ дополнительных вставок, поэтому к тому времени, когда в таблице будет m элементов и она станет заполненной, каждому элементу будет соответствовать доллар, предназначенный для оплаты его перемещения в новую таблицу в процессе расширения.

Анализ последовательности из n операций TABLE-INSERT можно выполнить и с помощью метода потенциалов. Этим мы сейчас и займемся; кроме того, этот метод будет использован в разделе 17.4.2 для разработки операции TABLE-DELETE, амортизированная стоимость которой равна $O(1)$. Начнем с того, что определим функцию потенциала Φ, которая становится равной 0 сразу после расширения и достигает значения, равного размеру матрицы, к тому времени, когда матрица станет заполненной. В этом случае предстоящее расширение можно будет оплатить за счет потенциала. Одним из возможных вариантов является функция

$$\Phi(T) = 2 \cdot T.num - T.size \; . \tag{17.5}$$

Сразу после расширения выполняется соотношение $T.num = T.size/2$, поэтому, как и требуется, $\Phi(T) = 0$. Непосредственно перед расширением справедливо равенство $T.num = T.size$, следовательно, как и требуется, $\Phi(T) = T.num$. Начальное значение потенциала равно 0, и, поскольку таблица всегда заполнена не менее чем наполовину, выполняется неравенство $T.num \geq T.size/2$, из которого следует, что функция $\Phi(T)$ всегда неотрицательна. Таким образом, суммарная амортизированная стоимость n операций TABLE-INSERT является верхней границей суммарной фактической стоимости.

Чтобы проанализировать амортизированную стоимость i-й операции TABLE-INSERT, обозначим через num_i количество элементов, хранящихся в таблице после этой операции, через $size_i$ — общий размер таблицы после этой операции и через Φ_i — потенциал после этой операции. Изначально $num_0 = 0$, $size_0 = 0$ и $\Phi_0 = 0$.

Если i-я операция TABLE-INSERT не приводит к расширению, то $size_i = size_{i-1}$ и амортизированная стоимость операции равна

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3 \; .
\end{aligned}$$

Если же i-я операция TABLE-INSERT приводит к расширению, то $size_i = 2 \cdot size_{i-1}$ и $size_{i-1} = num_{i-1} = num_i - 1$, откуда вытекает, что $size_i = 2 \cdot (num_i - 1)$. Таким

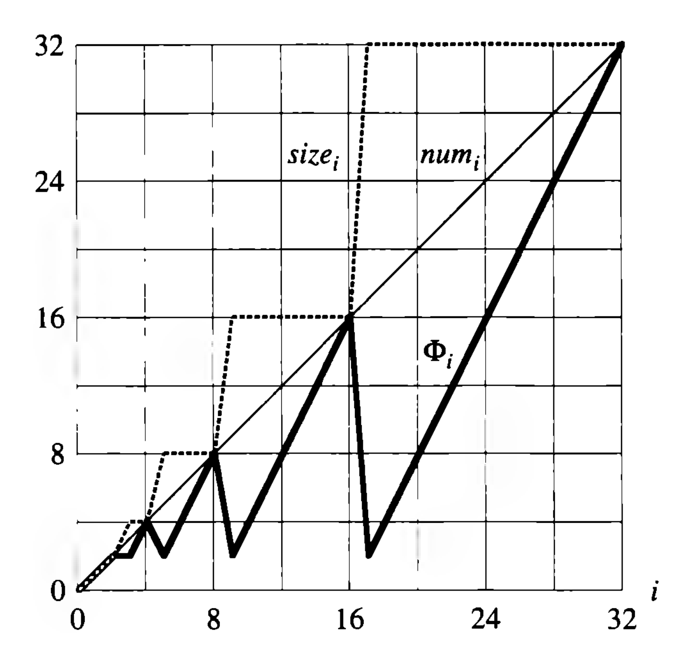


Рис. 17.3. Влияние последовательности n операций TABLE-INSERT на количество num_i элементов в таблице, количество $size_i$ ячеек таблицы и потенциал $\Phi_i = 2 \cdot num_i - size_i$, измеренные после каждой i-й операции. Величина num_i показана тонкой линией, величина $size_i$ — пунктирной линией, а Φ_i — толстой линией. Обратите внимание, что непосредственно перед расширением потенциал возрастает до значения, равного количеству содержащихся в таблице элементов, поэтому становится достаточным для оплаты перемещения всех элементов в новую таблицу. Впоследствии потенциал снижается до 0, но тут же возрастает до значения 2, когда в таблицу вставляется элемент, вызвавший ее расширение.

образом, амортизированная стоимость операции равна

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 \ .
\end{aligned}$$

На рис. 17.3 изображены зависимости значений num_i, $size_i$ и Φ_i от i. Обратите внимание на возрастание потенциала для оплаты расширения таблицы.

17.4.2. Расширение и сжатие таблицы

Чтобы реализовать операцию TABLE-DELETE, достаточно просто удалить из нее указанный элемент. Однако часто желательно ***сжать*** (contract) таблицу, если ее коэффициент заполнения становится слишком мал, чтобы пространство не расходовалось напрасно. Сжатие таблицы выполняется аналогично ее расширению: если количество элементов в ней достигает некоторой критической нижней отметки, выделяется место для новой, меньшей таблицы, после чего элементы из старой таблицы копируются в новую. Затем место, используемое для старой таблицы, можно будет освободить путем возвращения его системе управления памятью. В идеале желательно, чтобы выполнялись два условия:

- коэффициент заполнения динамической таблицы должен быть ограничен снизу некоторой положительной константой;
- амортизированная стоимость табличных операций должна быть ограничена сверху некоторой константой.

Предполагается, что эту стоимость можно измерить в терминах элементарных вставок и удалений.

Естественная стратегия при расширении и сжатии таблицы состоит в том, чтобы удваивать ее размер при вставке элемента в заполненную таблицу и в два раза уменьшать его, когда удаление элемента из таблицы приводит к тому, что она становится заполненной меньше чем наполовину. При этом гарантируется, что величина коэффициента заполнения таблицы не может быть меньше $1/2$. Однако, к сожалению, это может привести к тому, что амортизированная стоимость операции станет довольно большой. Рассмотрим такой сценарий. Над таблицей T выполняется n операций, где n — степень двойки. Первые $n/2$ операций — операции вставки, и их полная стоимость, согласно нашему предыдущему анализу, равна $\Theta(n)$. В конце этой последовательности вставок $T.num = T.size = n/2$. В качестве следующих $n/2$ операций выполним такую последовательность:

вставка, удаление, удаление, вставка, вставка, удаление, удаление, вставка, вставка, ...

Первая вставка приводит к расширению таблицы, объем которой равен n. Два следующих удаления вызовут сжатие таблицы обратно до размера $n/2$. Следующие две вставки приведут к еще одному расширению и т.д. Стоимость каждого расширения и сжатия равна $\Theta(n)$, а всего их — $\Theta(n)$. Таким образом, полная стоимость n операций равна $\Theta(n^2)$, а амортизированная стоимость одной операции — $\Theta(n)$.

Трудность, возникающая при использовании данной стратегии, очевидна: после расширения мы не успеваем выполнить достаточное количество удалений, чтобы оплатить сжатие. Аналогично после сжатия не выполняется достаточное количество вставок, чтобы оплатить расширение.

Эту стратегию можно улучшить, если позволить, чтобы коэффициент заполнения таблицы опускался ниже $1/2$. В частности, как и раньше, размер таблицы будет удваиваться при вставке элемента в заполненную таблицу, но ее размер будет уменьшаться в два раза, если в результате удаления таблица становится заполненной не наполовину, как это было раньше, а на четверть. При этом коэффициент заполнения таблицы будет ограничен снизу константой $1/4$.

Интуитивно мы рассматриваем коэффициент заполнения $1/2$ как идеальный, а потенциал таблицы при этом равен 0. При отклонении коэффициента заполнения от $1/2$ потенциал возрастает таким образом, чтобы к тому времени, когда мы будем расширять или сжимать таблицу, его было достаточно, чтобы оплатить копирование всех элементов во вновь выделенную таблицу. Таким образом, нам потребуется функция потенциала, которая растет до $T.num$, когда коэффициент заполнения либо возрастает до 1, либо уменьшается до $1/4$. После как сжатия, так и расширения таблицы коэффициент заполнения становится равным $1/2$, а потенциал таблицы обнуляется.

Код процедуры TABLE-DELETE здесь не приводится, поскольку он аналогичен коду процедуры TABLE-INSERT. Однако для анализа удобно предположить, что если количество элементов таблицы уменьшается до нуля, то выделенное для нее пространство освобождается, т.е. если $T.num = 0$, то $T.size = 0$.

Теперь можно проанализировать стоимость последовательности из n операций TABLE-INSERT и TABLE-DELETE, воспользовавшись методом потенциалов. Начнем с того, что определим функцию потенциала Φ, которая равна 0 сразу после расширения или сжатия и возрастает по мере того, как коэффициент заполнения увеличивается до 1 или уменьшается до 1/4. Обозначим коэффициент заполнения непустой таблицы T через $\alpha(T) = T.num/T.size$. Поскольку для пустой таблицы выполняются соотношения $T.num = T.size = 0$ и $\alpha(T) = 1$, равенство $T.num = \alpha(T) \cdot T.size$ всегда справедливо, независимо от того, является ли таблица пустой. В качестве функции потенциала будет использоваться следующая:

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size \;, & \text{если } \alpha(T) \geq 1/2 \;, \\ T.size/2 - T.num \;, & \text{если } \alpha(T) < 1/2 \;. \end{cases} \tag{17.6}$$

Обратите внимание, что потенциал пустой таблицы равен 0 и что функция потенциала ни при каких аргументах не принимает отрицательного значения. Таким образом, полная амортизированная стоимость последовательности операций для функции Φ является верхней границей фактической стоимости этой последовательности.

Прежде чем продолжить точный анализ, сделаем паузу для обсуждения некоторых свойств функции потенциала, проиллюстрированной на рис. 17.4. Заметим, что когда коэффициент заполнения равен 1/2, потенциал равен 0. Когда коэффициент заполнения равен 1, выполняется соотношение $T.size = T.num$, откуда вытекает $\Phi(T) = T.num$, следовательно, значение потенциала оказывается достаточным для оплаты расширения в случае добавления элемента. Если же коэффициент заполнения равен 1/4, выполняется соотношение $T.size = 4 \cdot T.num$, откуда следует $\Phi(T) = T.num$, и в этом случае значение потенциала также является достаточным для оплаты сжатия в результате удаления элемента.

Чтобы проанализировать последовательность n операций TABLE-INSERT и TABLE-DELETE, введем следующие обозначения: обозначим через c_i фактическую стоимость i-й операции, через $\widehat{c}_i$ — ее амортизированную стоимость в соответствии с функцией Φ, через num_i — количество хранящихся в таблице элементов после выполнения i-й операции, через $size_i$ — полный объем таблицы после выполнения i-й операции, через α_i — коэффициент заполнения таблицы после выполнения i-й операции и через Φ_i — потенциал после выполнения i-й операции. Изначально $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$ и $\Phi_0 = 0$.

Начнем со случая, когда i-й по счету выполняется операция TABLE-INSERT. Если $\alpha_{i-1} \geq 1/2$, то этот анализ идентичен тому, который был проведен в разделе 17.4.1 для расширения таблицы. Независимо от того, расширяется таблица или нет, амортизированная стоимость $\widehat{c}_i$ операции не превышает 3. Если $\alpha_{i-1} < 1/2$, таблица не может быть расширена в результате выполнения операции, поскольку расширение происходит только тогда, когда $\alpha_{i-1} = 1$. При $\alpha_i < 1/2$ амортизиро-

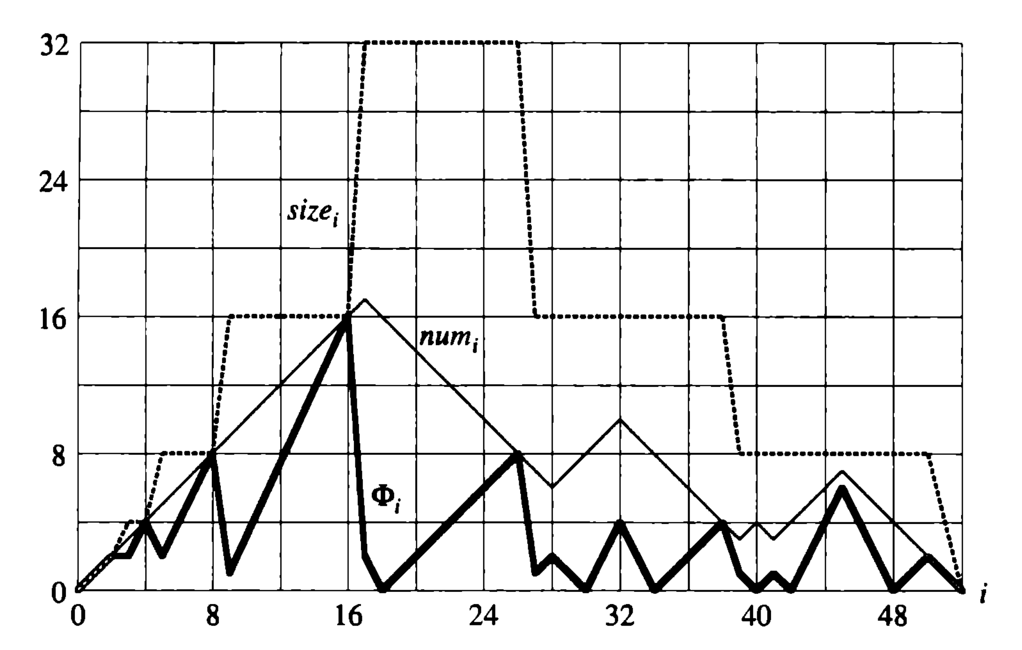


Рис. 17.4. Влияние последовательности n операций TABLE-INSERT и TABLE-DELETE на количество num_i элементов в таблице, количество $size_i$ ячеек таблицы и потенциал

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i \ , & \text{если } \alpha_i \geq 1/2 \ , \\ size_i/2 - num_i \ , & \text{если } \alpha_i < 1/2 \ , \end{cases}$$

измеренные после каждой i-й операции. Величина num_i показана тонкой линией, $size_i$ — пунктирной, а Φ_i — толстой линией. Обратите внимание, что непосредственно перед расширением потенциал возрастает до значения, равного количеству содержащихся в таблице элементов, поэтому становится достаточным для оплаты перемещения всех элементов в новую таблицу. Аналогично непосредственно перед сжатием потенциал также возрастает до значения, равного количеству содержащихся в таблице элементов.

ванная стоимость i-й операции равна

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0 \ .
\end{aligned}$$

Если $\alpha_{i-1} < 1/2$, но $\alpha_i \geq 1/2$, имеем

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3 \ .
\end{aligned}$$

Таким образом, амортизированная стоимость операции TABLE-INSERT не превышает 3.

Теперь обратимся к случаю, когда i-й по счету выполняется операция TABLE-DELETE. В этом случае $num_i = num_{i-1} - 1$. Если $\alpha_{i-1} < 1/2$, то необходимо анализировать, приведет ли эта операция к сжатию матрицы. Если нет, выполняется условие $size_i = size_{i-1}$, и амортизированная стоимость операции равна

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\
&= 2 \,.
\end{aligned}$$

Если $\alpha_{i-1} < 1/2$ и i-я операция приводит к сжатию, то фактическая стоимость операции равна $c_i = num_i + 1$, поскольку один элемент удаляется и num_i элементов перемещается. В этом случае $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$, и амортизированная стоимость операции равна

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\
&= 1 \,.
\end{aligned}$$

Если i-я операция — TABLE-DELETE и выполняется неравенство $\alpha_{i-1} \geq 1/2$, то амортизированная стоимость также ограничена сверху константой. Анализ этого случая предлагается выполнить в упр. 17.4.2.

Подведем итог. Поскольку амортизированная стоимость каждой операции ограничена сверху константой, фактическая стоимость выполнения произвольной последовательности из n операций над динамической таблицей равна $O(n)$.

Упражнения

17.4.1

Предположим, что нужно реализовать динамическую хеш-таблицу с открытой адресацией. Почему может понадобиться считать такую таблицу заполненной, когда ее коэффициент заполнения достигнет некоторого значения α, строго меньшего 1? Кратко опишите, как следует выполнять вставки в динамическую хеш-таблицу с открытой адресацией, чтобы математическое ожидание амортизированной стоимости вставки было равно $O(1)$. Почему математическое ожидание фактической стоимости вставки может быть равным $O(1)$ не для всех вставок?

17.4.2

Покажите, что если $\alpha_{i-1} \geq 1/2$ и i-я операция, выполняющаяся над динамической таблицей, — TABLE-DELETE, то амортизированная стоимость операции для функции потенциала (17.6) ограничена сверху константой.

17.4.3

Предположим, что вместо того, чтобы сжимать таблицу, вдвое уменьшая ее размер при уменьшении коэффициента заполнения ниже значения $1/4$, мы будем сжимать ее до $2/3$ исходного размера, когда значение коэффициента заполнения становится меньше $1/3$. С помощью функции потенциала

$$\Phi(T) = |2 \cdot T.num - T.size|$$

покажите, что амортизированная стоимость процедуры TABLE-DELETE, в которой применяется эта стратегия, ограничена сверху константой.

Задачи

17.1. Обратный бинарный битовый счетчик

В главе 30 описан важный алгоритм, называемый быстрым преобразованием Фурье (Fast Fourier Transform — FFT). На первом этапе алгоритма FFT выполняется ***обратная перестановка битов*** (bit-reversal permutation) входного массива $A[0..n-1]$, длина которого равна $n = 2^k$ для некоторого неотрицательного целого k. В результате перестановки каждый элемент массива меняется местом с элементом, индекс которого является числом с бинарным представлением, обратным по отношению к бинарному представлению исходного индекса.

Каждый индекс a можно представить в виде последовательности k бит $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$, где $a = \sum_{i=0}^{k-1} a_i 2^i$. Определим операцию

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle ;$$

таким образом,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

Например, если $n = 16$ (или, что эквивалентно, $k = 4$), то $\text{rev}_k(3) = 12$, поскольку 4-битовое представление 3 — 0011, которое при обращении дает 1100, 4-битовое представление 12.

а. Пусть имеется функция rev_k, выполняющаяся в течение времени $\Theta(k)$. Разработайте алгоритм обратной перестановки битов в массиве длиной $n = 2^k$, время работы которого равно $O(nk)$.

Чтобы улучшить время перестановки битов, можно воспользоваться алгоритмом, основанным на амортизационном анализе. Мы поддерживаем счетчик с обратным порядком битов и процедуру BIT-REVERSED-INCREMENT, которая для заданного показания счетчика a возвращает значение $\text{rev}_k(\text{rev}_k(a) + 1)$. Например, если $k = 4$ и счетчик начинает отсчет от 0, то в результате последовательных вызовов

процедуры BIT-REVERSED-INCREMENT получим последовательность

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots \ .$$

б. Предположим, что слова в компьютере хранятся в виде k-битовых значений и что в течение единичного промежутка времени компьютер может произвести над этими бинарными значениями такие операции, как сдвиг влево или вправо на произвольное число позиций, побитовое И, побитовое ИЛИ и т.п. Опишите реализацию процедуры BIT-REVERSED-INCREMENT, позволяющую выполнять обращение порядка битов n-элементного массива за время $O(n)$.

в. Предположим, что в течение единичного промежутка времени слово можно сдвинуть вправо или влево только на один бит. Сохраняется ли при этом возможность реализовать обращение порядка битов в массиве за время $O(n)$?

17.2. Динамический бинарный поиск

Бинарный поиск в отсортированном массиве осуществляется в течение промежутка времени, описываемого логарифмической функцией, однако время вставки нового элемента линейно зависит от размера массива. Время вставки можно улучшить, используя несколько отсортированных массивов.

В частности, предположим, что операции SEARCH и INSERT должны поддерживаться для n-элементного множества. Пусть $k = \lceil \lg(n+1) \rceil$ и пусть бинарное представление числа n имеет вид $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. У нас имеется k отсортированных массивов $A_0, A_1, \ldots, A_{k-1}$, где для всех $i = 0, 1, \ldots, k-1$ длина массива A_i равна 2^i. Каждый массив либо заполнен, либо пуст, в зависимости от того, равно ли $n_i = 1$ или $n_i = 0$ соответственно. Таким образом, полное количество элементов, хранящихся во всех k массивах, равно $\sum_{i=0}^{k-1} n_i\, 2^i = n$. Хотя каждый массив отсортирован, между элементами различных массивов нет никаких отношений.

а. Опишите, как реализовать операцию SEARCH в этой структуре данных. Проанализируйте время ее работы в наихудшем случае.

б. Опишите, как вставлять новый элемент в эту структуру данных. Проанализируйте время выполнения этой операции INSERT в наихудшем случае и ее амортизированное время работы.

в. Подумайте, как реализовать операцию DELETE.

17.3. Амортизированные сбалансированные по весу деревья

Рассмотрим обычное бинарное дерево поиска, расширенное за счет добавления к каждому узлу x атрибута $x.size$, содержащего количество ключей, которые хранятся в поддереве с корнем в узле x. Пусть α — константа из диапазона $1/2 \leq \alpha < 1$. Назовем данный узел x ***α-сбалансированным*** (α-balanced), если $x.left.size \leq \alpha \cdot x.size$ и $x.right.size \leq \alpha \cdot x.size$. Дерево в целом является ***α-сбалансированным***, если α-сбалансирован каждый его узел. Описанный ниже

амортизированный подход к сбалансированным деревьям был предложен Д. Варгисом (G. Varghese).

а. В определенном смысле $1/2$-сбалансированное дерево является наиболее сбалансированным. Пусть в произвольном бинарном дереве поиска задан узел x. Покажите, как перестроить поддерево с корнем в узле x таким образом, чтобы оно стало $1/2$-сбалансированным. Время работы алгоритма должно быть равным $\Theta(x.size)$, а объем используемой вспомогательной памяти — $O(x.size)$.

б. Покажите, что поиск в бинарном α-сбалансированном дереве поиска с n узлами в наихудшем случае выполняется за время $O(\lg n)$.

При выполнении оставшейся части этой задачи предполагается, что константа α строго больше $1/2$. Предположим, что операции INSERT и DELETE реализованы так же, как и в обычном бинарном дереве поиска с n узлами, за исключением того, что после каждой такой операции, если хотя бы один узел дерева не является α-сбалансированным, поддерево с самым высоким несбалансированным узлом "перестраивается" так, чтобы оно стало $1/2$-сбалансированным.

Проанализируем эту схему с помощью метода потенциалов. Для узла x, принадлежащего бинарному дереву поиска T, определим величину

$$\Delta(x) = |x.left.size - x.right.size| ,$$

а потенциал дерева T определим как

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) ,$$

где c — достаточно большая константа, зависящая от α.

в. Докажите, что потенциал любого бинарного дерева поиска всегда неотрицателен и что потенциал $1/2$-сбалансированного дерева равен 0.

г. Предположим, что m единиц потенциала достаточно для оплаты перестройки поддерева, содержащего m узлов. Насколько большой должна быть константа c, выраженная через α, чтобы амортизированное время перестройки поддерева, не являющегося α-сбалансированным, было равным $O(1)$?

д. Покажите, что амортизированное время вставки узла в α-сбалансированное дерево с n узлами, как и удаления узла из этого дерева, равно $O(\lg n)$.

17.4. Стоимость перестройки красно-черных деревьев

Имеется четыре основные операции над красно-черными деревьями, которые выполняют ***структурные модификации*** (structural modifications): вставка узла, удаление узла, поворот и изменение цвета. Мы видели, что для поддержки свойств красно-черных деревьев в процедурах RB-INSERT и RB-DELETE достаточно использовать $O(1)$ операций поворота, вставки узлов и удаления узлов, но в них требуется намного больше операций изменения цвета.

а. Опишите корректное красно-черное дерево с n узлами, в котором вызов процедуры RB-INSERT для вставки $(n+1)$-го узла приводит к $\Omega(\lg n)$ изменениям цвета. Затем опишите корректное красно-черное дерево с n узлами, в котором к $\Omega(\lg n)$ изменениям цвета приводит вызов процедуры RB-DELETE для удаления определенного узла.

Несмотря на то что в наихудшем случае количество изменений цветов, приходящихся на одну операцию, может оказаться логарифмическим, мы докажем, что произвольная последовательность m операций RB-INSERT и RB-DELETE, выполняемых над изначально пустым красно-черном деревом, в наихудшем случае приводит к $O(m)$ структурным изменениям.

б. Некоторые из случаев, обрабатываемых в основном цикле кода процедур RB-INSERT-FIXUP и RB-DELETE-FIXUP, являются ***завершающими*** (terminating): встретившись, они вызовут завершение цикла после выполнения фиксированного количества дополнительных операций. Установите для каждого случая, встречающегося в процедурах RB-INSERT-FIXUP и RB-DELETE-FIXUP, какой из них является завершающим, а какой — нет. (*Указание:* обратитесь к рис. 13.5–13.7.)

Сначала проанализируем структурные модификации для случая, когда выполняются только вставки. Пусть T — красно-черное дерево. Определим для него функцию $\Phi(T)$ как количество красных узлов в дереве T. Предположим, что одной единицы потенциала достаточно для оплаты структурных изменений, выполняемых в любом из трех случаев процедуры RB-INSERT-FIXUP.

в. Пусть T' — результат применения случая 1 из процедуры RB-INSERT-FIXUP к T. Докажите, что $\Phi(T') = \Phi(T) - 1$.

г. Операцию вставки узла в красно-черное дерево с помощью процедуры RB-INSERT можно разбить на три части. Перечислите структурные модификации и изменения потенциала, которые происходят в результате выполнения строк 1–16 процедуры RB-INSERT для незавершающих случаев процедуры RB-INSERT-FIXUP и для завершающих случаев этой процедуры.

д. Воспользовавшись результатами, полученными в п. (г), докажите, что амортизированное количество структурных модификаций, которые выполняются при любом вызове процедуры RB-INSERT, равно $O(1)$.

Теперь нам нужно доказать, что если выполняются и вставки, и удаления, то выполняются $O(m)$ структурных модификаций. Для каждого узла x определим величину

$$w(x) = \begin{cases} 0\ , & \text{если } x \text{ красный}\ , \\ 1\ , & \text{если } x \text{ черный и не имеет красных дочерних узлов}\ , \\ 0\ , & \text{если } x \text{ черный и имеет один красный дочерний узел}\ , \\ 2\ , & \text{если } x \text{ черный и имеет два красных дочерних узла}\ . \end{cases}$$

Теперь переопределим потенциал красно-черного дерева T как

$$\Phi(T) = \sum_{x \in T} w(x)$$

и обозначим через T' дерево, получаемое в результате применения любого незавершающего случая процедуры RB-INSERT-FIXUP или RB-DELETE-FIXUP к дереву T.

е. Покажите, что для всех незавершающих случаев процедуры RB-INSERTFIXUP выполняется неравенство $\Phi(T') \leq \Phi(T) - 1$. Докажите, что амортизированное количество структурных модификаций, которые выполняются в произвольном вызове процедуры RB-INSERT-FIXUP, равно $O(1)$.

ж. Покажите, что для всех незавершающих случаев процедуры RB-DELETE-FIXUP выполняется неравенство $\Phi(T') \leq \Phi(T) - 1$. Докажите, что амортизированное количество структурных модификаций, которые выполняются в произвольном вызове процедуры RB-DELETE-FIXUP, равно $O(1)$.

з. Завершите доказательство того, что в наихудшем случае в ходе выполнения произвольной последовательности m операций RB-INSERT и RB-DELETE выполняется $O(m)$ структурных модификаций.

17.5. Конкурентный анализ самоорганизующихся списков с перемещением в начало

Самоорганизующийся список (self-organizing list) представляет собой связанный список из n элементов, в котором каждый элемент имеет уникальный ключ. Поиск в списке заключается в поиске элемента с заданным ключом.

Самоорганизующийся список обладает двумя важными свойствами.

1. Для поиска в списке элемента с заданным ключом требуется пройти по списку от его начала до тех пор, пока не встретится элемент с искомым ключом. Если это k-й от начала списка элемент, то стоимость его поиска равна k.
2. Можно переупорядочивать элементы списка после любой операции в соответствии с некоторым правилом и определенной стоимостью. Можно выбрать любую эвристику для принятия решения о том, как именно переупорядочивать список.

Предположим, что мы начинаем работу с заданного списка с n элементами, и при этом задана последовательность $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$ искомых ключей (поиск выполняется в указанном порядке). Стоимостью последовательности является сумма стоимостей отдельных поисков в последовательности.

Помимо различных возможных способов переупорядочения списка после операции, в данной задаче рассматривается обмен местами соседних элементов списка (с единичной стоимостью каждой такой операции). Применив функцию потенциала, необходимо показать, что определенная эвристика переупорядочения

17 Зак. 3726

списка (в данном случае — перемещение элемента в начало) имеет общую стоимость, не более чем в 4 раза превышающую стоимость любой иной эвристики для поддержания упорядочения списка — даже если этой эвристике заранее известна последовательность поисков! Этот тип анализа называется ***конкурентным анализом***.

Обозначим для заданной эвристики H и заданного начального упорядочения списка стоимость последовательности поисков σ как $C_H(\sigma)$. Пусть m — количество поисков в последовательности σ.

а. Докажите, что если эвристике H список поиска заранее не известен, то стоимость эвристики H для последовательности σ в наихудшем случае равна $C_H(\sigma) = \Omega(mn)$.

В случае эвристики ***перемещения в начало*** непосредственно после того, как найден искомый элемент x, он перемещается в первую позицию списка (т.е. в его начало).

Обозначим ранг элемента x в списке L (т.е. его позицию в списке) через $\mathrm{rank}_L(x)$. Например, если x является четвертым элементом в L, то $\mathrm{rank}_L(x) = 4$. Обозначим через c_i стоимость поиска σ_i с применением эвристики перемещения в начало, которая включает стоимость поиска элемента в списке и стоимость его перемещения в начало списка путем ряда обменов соседних элементов списка.

б. Покажите, что если σ_i находит элемент x в списке L с применением эвристики перемещения в начало, то $c_i = 2 \cdot \mathrm{rank}_L(x) - 1$.

Теперь сравним перемещение в начало с любой другой эвристикой H, которая обрабатывает последовательность поисков в соответствии со сформулированными выше свойствами. Эвристика H может обменивать элементы в списке любым способом и даже может знать всю последовательность поисков заранее.

Пусть L_i — список после поиска σ_i с использованием перемещения в начало, а L_i^* — список после поиска σ_i с использованием эвристики H. Обозначим стоимость поиска σ_i как c_i в случае перемещения в начало, и как c_i^* — в случае эвристики H. Предположим, что эвристика H выполняет t_i^* обменов в процессе поиска σ_i.

в. В п. (б) вы показали, что $c_i = 2 \cdot \mathrm{rank}_{L_{i-1}}(x) - 1$. Теперь покажите, что $c_i^* = \mathrm{rank}_{L_{i-1}^*}(x) + t_i^*$.

Определим ***инверсию*** в списке L_i как пару элементов y и z, такую, что y предшествует z в L_i, а z предшествует y в списке L_i^*. Предположим, что после обработки последовательности поисков $\langle \sigma_1, \sigma_2, \ldots, \sigma_i \rangle$ список L_i имеет q_i инверсий. Тогда мы определим функцию потенциала Φ, которая отображает L_i на действительное число следующим образом: $\Phi(L_i) = 2q_i$. Например, если L_i имеет элементы $\langle e, c, a, d, b \rangle$, а L_i^* имеет элементы $\langle c, a, b, d, e \rangle$, то L_i имеет 5 инверсий $((e,c), (e,a), (e,d), (e,b), (d,b))$, так что $\Phi(L_i) = 10$. Заметим, что $\Phi(L_i) \geq 0$ для всех i и что, если эвристика перемещения в начало и эвристика H начинают работу с одного и того же списка L_0, $\Phi(L_0) = 0$.

г. Докажите, что обмены либо увеличивают потенциал на 2, либо уменьшают на 2.

Предположим, что в результате поиска σ_i найден элемент x. Чтобы понять, как в результате поиска σ_i изменится потенциал, разделим элементы, отличные от x, на четыре множества, в зависимости от того, где они находятся в списке непосредственно перед i-м поиском.

- Множество A состоит из элементов, предшествующих x как в L_{i-1}, так и в L^*_{i-1}.
- Множество B состоит из элементов, предшествующих x в L_{i-1} и следующих за x в L^*_{i-1}.
- Множество C состоит из элементов, следующих за x в L_{i-1} и предшествующих ему в L^*_{i-1}.
- Множество D состоит из элементов, следующих за x как в L_{i-1}, так и в L^*_{i-1}.

д. Докажите, что $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ и $\text{rank}_{L^*_{i-1}}(x) = |A| + |C| + 1$.

е. Покажите, что поиск σ_i приводит к изменению потенциала

$$\Phi(L_i) - \Phi(L_{i-1}) \le 2(|A| - |B| + t^*_i) ,$$

где, как и ранее, эвристика H выполняет t^*_i обменов в процессе поиска σ_i.

Определим амортизированную стоимость $\widehat{c}_i$ поиска σ_i как $\widehat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

ж. Покажите, что амортизированная стоимость $\widehat{c}_i$ поиска σ_i ограничена сверху величиной $4c^*_i$.

з. Сделайте вывод о том, что стоимость $C_{\text{MTF}}(\sigma)$ последовательности поисков σ с перемещением в начало превышает стоимость $C_H(\sigma)$ последовательности σ с применением любой другой эвристики H не более чем в 4 раза, в предположении, что обе эвристики начинают работу с одним и тем же списком.

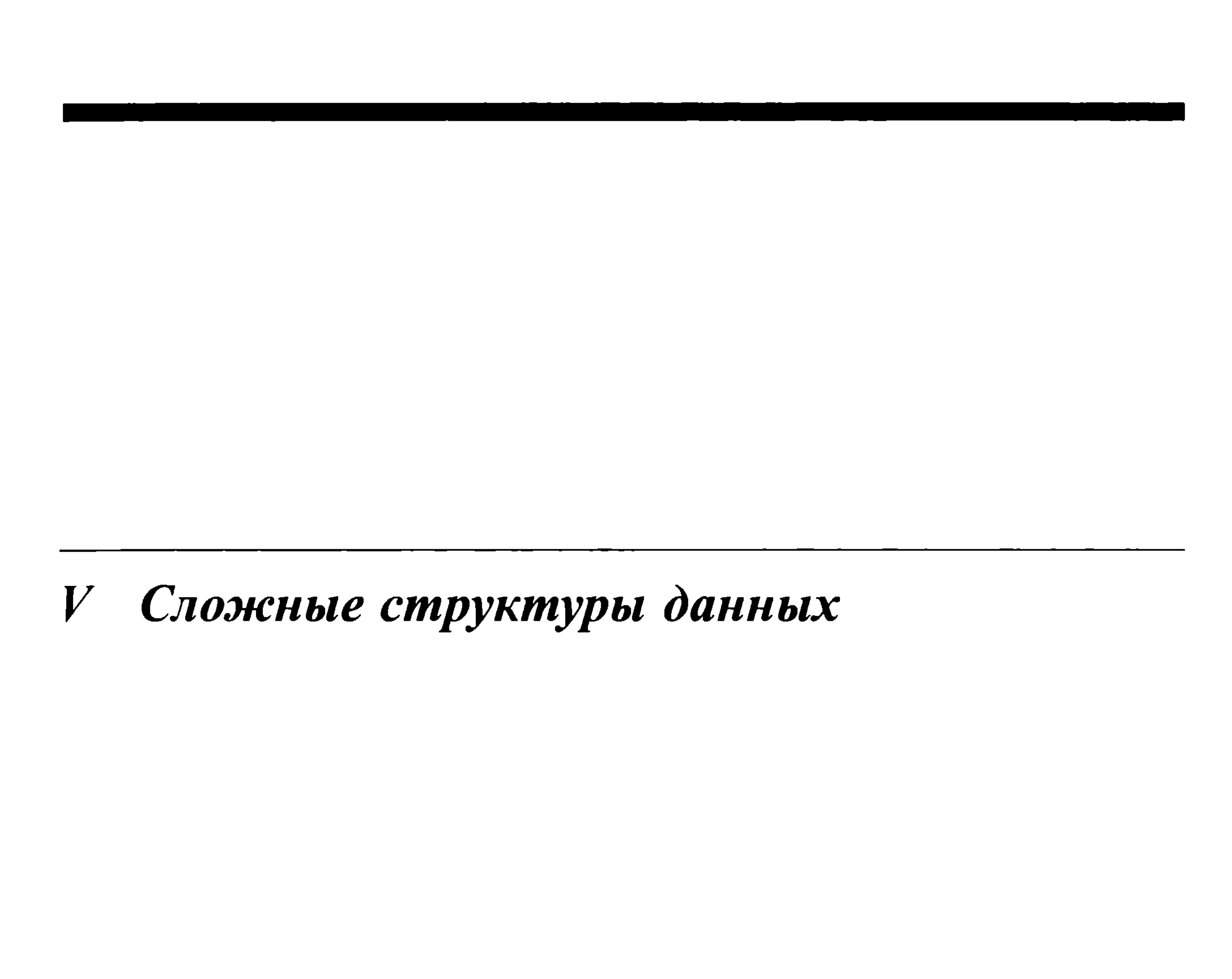
Заключительные замечания

Групповой анализ был использован Ахо (Aho), Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5] для определения времени работы операций над лесом непересекающихся множеств; мы будем анализировать эту структуру данных с применением метода потенциалов в главе 21. Таржан (Tarjan) [329] представил обзор разновидностей метода бухгалтерского учета и метода потенциала, использующихся в амортизационном анализе, а также некоторые их приложения. Метод бухгалтерского учета он приписывает нескольким авторам, включая М. Брауна (M. Brown), Таржана, С. Хаддельстона (S. Huddelston) и К. Мельхорна (K. Mehlhorn). Метод

потенциала он приписывает Д.Д. Слитору (D.D. Sleator). Термин "амортизационный" вошел в обиход благодаря Слитору и Таржану.

Функции потенциалов оказываются полезными также при доказательстве нижних границ в задачах определенных типов. Для каждой конфигурации, возникающей в такой задаче, определяется функция потенциала, отображающая эту конфигурацию на действительное число. После этого определяется потенциал Φ_{init} начальной конфигурации, потенциал Φ_{final} конечной конфигурации и максимальное изменение потенциала $\Delta\Phi_{\text{max}}$ в результате выполнения произвольного шага. Таким образом, число шагов должно быть не меньше $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. Примеры использования функций потенциалов для обоснования нижних границ при оценке сложности ввода-вывода представлены в работах Кормена (Cormen), Сандквиста (Sundquist) и Вишневски (Wisniewski) [78], Флойда (Floyd) [106], а также Аггарвала (Aggarwal) и Виттера (Vitter) [3]. Крумм (Krumme), Цыбенко (Cybenko) и Венкатараман (Venkataraman) [220] воспользовались потенциальными функциями для оценки нижних границ ***скорости распространения слухов*** (gossiping): передачи единственного элемента из каждой вершины графа во все другие вершины.

Эвристика перемещения в начало из задачи 17.5 достаточно хорошо работает на практике. Более того, если при обнаружении элемента мы можем поместить его в начало списка за константное время, то можно показать, что стоимость эвристики перемещения в начало превышает стоимость любой иной эвристики не более чем в два раза, даже если этой эвристике весь список поисков известен заранее.



V Сложные структуры данных

Введение

В этой части мы возвращаемся к изучению структур данных, поддерживающих операции над динамическими множествами, но уже на более высоком уровне, чем в части III. В двух первых главах этой части мы, например, будем использовать технологии амортизационного анализа, которые рассматриваются в главе 17.

В главе 18 мы рассмотрим В-деревья, которые представляют собой сбалансированные деревья поиска, разработанные специально для хранения информации на дисках. Поскольку дисковые операции существенно медленнее, чем работа с оперативной памятью, производительность В-деревьев определяется не только временем, необходимым для проведения вычислений, но и количеством необходимых дисковых операций. Для каждой операции с В-деревом количество обращений к диску растет с ростом высоты В-дерева, так что мы должны поддерживать ее небольшой.

В главе 19 рассматривается реализация объединяемых пирамид (mergeable heap), которые поддерживают операции INSERT, MINIMUM, EXTRACT-MIN и UNION.[1] Процедура UNION сливает, или объединяет, две пирамиды. Пирамиды Фибоначчи — структуры данных из главы 19 — также поддерживают операции DELETE и DECREASE-KEY. Для определения производительности фибоначчиевых пирамид мы используем амортизированные границы времени выполнения операций. Операции INSERT, MINIMUM и UNION в пирамидах Фибоначчи обладают фактическим и амортизированным временем выполнения, равным $O(1)$, а операции EXTRACT-MIN и DELETE обладают амортизированным временем работы $O(\lg n)$. Однако основное преимущество пирамид Фибоначчи в том, что

[1]Как и в задаче 10.2, мы определяем пирамиды как поддерживающие операции MINIMUM и EXTRACT-MIN, так что их можно называть ***объединяемыми неубывающими пирамидами***. Если пирамидой поддерживаются операции MAXIMUM и EXTRACT-MAX, то такая пирамида является ***объединяемой невозрастающей пирамидой***. Если не указано иное, по умолчанию под объединяемыми пирамидами мы будем подразумевать объединяемые неубывающие пирамиды.

амортизированное время работы процедуры DECREASE-KEY составляет всего лишь $O(1)$. Именно это свойство делает фибоначчиевы пирамиды ключевым компонентом ряда наиболее быстрых асимптотических алгоритмов для работы с графами.

С учетом того, что нижняя граница времени сортировки $\Omega(n \lg n)$ может быть превзойдена в случае, когда ключи представляют собой целые числа из ограниченного диапазона, в главе 20 выясняется, можно ли разработать структуру данных, которая поддерживает операции над динамическим множеством SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR со временем работы $o(\lg n)$ для ключей, являющихся целочисленными значениями из ограниченного диапазона. Оказывается, что ответ на этот вопрос положительный, если воспользоваться рекурсивной структурой данных, известной как дерево ван Эмде Боаса (van Emde Boas). Если ключи представляют собой уникальные целые числа из множества $\{0, 1, 2, \ldots, u-1\}$, где u является точной степенью 2, то деревья ван Эмде Боаса поддерживают выполнение всех перечисленных выше операций за время $O(\lg \lg u)$.

И наконец в главе 21 представлены структуры данных для хранения непересекающихся множеств. У нас имеется универсум из n элементов, сгруппированных в динамические множества. Изначально каждый элемент принадлежит своему собственному множеству. Операция UNION позволяет объединить два множества, а запрос FIND-SET определяет множество, в котором в данный момент находится искомый элемент. Представляя каждое множество в виде простого корневого дерева, мы получаем возможность удивительно производительной работы: последовательность из m операций выполняется за время $O(m\,\alpha(n))$, где $\alpha(n)$ — исключительно медленно растущая функция и не превышает 4 для всех мыслимых приложений. Амортизационный анализ, который позволяет доказать эту оценку, столь же сложен, сколь проста сама используемая структура.

Рассмотренные в этой части книги структуры данных — не более чем примеры "сложных" структур данных, и множество интересных структур сюда не вошли. Хотелось бы упомянуть следующие из них.

- ***Динамические деревья*** (dynamic trees), разработанные Слитором (Sleator) и Таржаном (Tarjan) [317] и детально рассмотренные в [328], поддерживают лес из непересекающихся корневых деревьев. Каждое ребро каждого дерева имеет некоторую стоимость, выраженную действительным числом. Динамические деревья поддерживают запросы поиска родителя, корня, стоимости ребра и минимальной стоимости ребра на пути от узла к корню. У деревьев могут удаляться ребра, может изменяться стоимость на простом пути от узла к корню, корень может связываться с другим деревом, можно также делать узел корнем дерева, в котором он находится. Имеется реализация динамических деревьев, которая обеспечивает амортизированное время работы каждой операции, равное $O(\lg n)$; другая, более сложная, реализация обеспечивает время работы $O(\lg n)$ в наихудшем случае.
- ***Расширяющиеся деревья*** (splay trees), также разработанные Слитором (Sleator) и Таржаном (Tarjan) [318] и также детально описанные в упоми-

навшейся выше работе [328], представляют собой версию бинарных деревьев поиска, в которых операции поиска имеют амортизированное время работы $O(\lg n)$. Одно из применений расширяющихся деревьев — упрощение реализации динамических деревьев.

- ***Перманентные структуры*** (persistent data structures) обеспечивают возможность выполнения запросов о предшествующих версиях структуры данных, а иногда и их обновления. Соответствующие методы с малыми затратами времени и памяти были предложены в работе [96] Дрисколлом (Driscoll), Сарнаком (Sarnak), Слитором (Sleator) и Таржаном (Tarjan). В задаче 13.1 приведен простой пример перманентного динамического множества.
- Как и рассматриваемые в главе 20, ряд структур данных обеспечивает более быстрое выполнение словарных операций (INSERT, DELETE, SEARCH) для ограниченных универсумов ключей. Такие ограничения позволяют получить лучшее асимптотическое время работы, чем для структур данных, основанных на сравнении. Фредман (Fredman) и Виллард (Willard) [114] предложили использовать ***деревья слияний*** (fusion trees), которые были первыми структурами данных, обеспечивавшими повышение скорости словарных операций за счет ограничения пространства целыми числами. Они показали, как можно реализовать эти операции, чтобы их время работы составляло $O(\lg n / \lg \lg n)$. Ряд других структур данных, включая ***экспоненциальные деревья поиска*** (exponential search trees) [16], также обеспечивают улучшение всех (или некоторых) словарных операций и упоминаются в заключительных замечаниях в некоторых главах книги.
- ***Динамические графы*** (dynamic graph) поддерживают различные запросы, позволяя структуре графа изменяться в процессе операций добавления или удаления вершин или ребер. Примеры поддерживаемых запросов включают связность вершин [165], связность ребер, минимальные остовные деревья [164], двусвязность и транзитивное замыкание [163].

В заключительных замечаниях к различным главам данной книги упоминаются, помимо описанных, некоторые другие структуры данных.

Глава 18. В-деревья

В-деревья представляют собой сбалансированные деревья поиска, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом). В-деревья похожи на красно-черные деревья (см. главу 13), но отличаются более высокой оптимизацией дисковых операций ввода-вывода. Многие СУБД используют для хранения информации именно В-деревья или их разновидности.

В-деревья отличаются от красно-черных деревьев тем, что узлы В-дерева могут иметь много дочерних узлов — от нескольких штук до тысяч, так что степень ветвления В-дерева может быть очень большой (хотя обычно она определяется характеристиками используемых дисков). В-деревья схожи с красно-черными деревьями в том, что все В-деревья с n узлами имеют высоту $O(\lg n)$, хотя само значение высоты В-дерева существенно меньше, чем у красно-черного дерева за счет более сильного ветвления. Таким образом, В-деревья также могут использоваться для реализации многих операций над динамическими множествами за время $O(\lg n)$.

В-деревья представляют собой естественное обобщение бинарных деревьев поиска. На рис. 18.1 показан пример простого В-дерева. Если внутренний узел x В-дерева содержит $x.n$ ключей, то у него $x.n + 1$ дочерних узлов. Ключи в узле x используются как разделители диапазона ключей, с которыми имеет дело данный узел, на $x.n + 1$ поддиапазонов, каждый из которых относится к одному из дочерних узлов x. При поиске ключа в В-дереве мы выбираем один из $(x.n + 1)$ дочерних узлов путем сравнения искомого значения с $x.n$ ключами, хранящимися в узле x. Структура листьев В-дерева отличается от структуры внутренних узлов; мы рассмотрим эти отличия в разделе 18.1.

В разделе 18.1 приведено точное определение В-деревьев и доказан логарифмический рост высоты В-дерева в зависимости от количества его узлов. В разделе 18.2 описаны процессы поиска в В-дереве и вставки элемента в В-дерево, а в разделе 18.3 рассматривается процесс удаления из В-дерева. Однако перед тем как приступить к работе с В-деревьями, давайте выясним, почему структуры данных, созданные для работы с дисковой памятью, так отличаются от структур для работы с оперативной памятью.

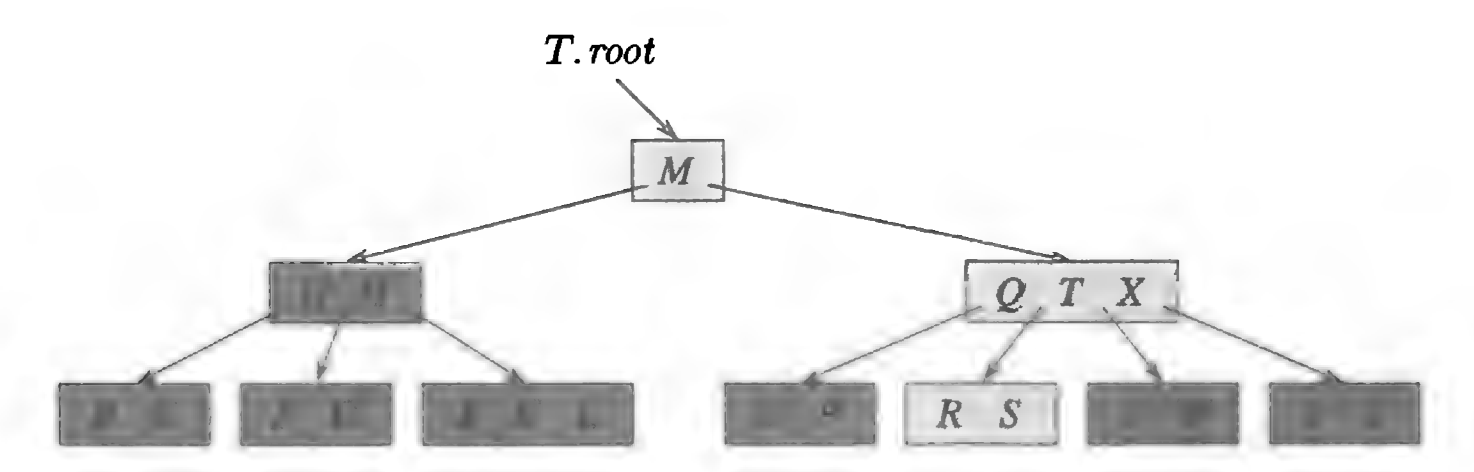


Рис. 18.1. В-дерево с согласными английского алфавита в качестве ключей. Внутренний узел x, содержащий $x.n$ ключей, имеет $x.n + 1$ дочерних узлов. Все листья находятся в дереве на одной и той же глубине. Светлой штриховкой показаны узлы, исследуемые при поиске буквы R.



Рис. 18.2. Типичный дисковый накопитель. Он состоит из одного или нескольких дисков (на рисунке их два), вращающихся на шпинделе. Каждый диск считывается и записывается с помощью головки на конце рычага. Рычаги вращаются на общей опорной оси. Дорожка представляет собой поверхность диска, проходящую под головкой, когда та находится в зафиксированном положении.

Структуры данных во вторичной памяти

Имеется несколько видов используемой компьютером памяти. ***Основная***, или ***оперативная***, ***память*** (primary, main memory) представляет собой специализированные микросхемы и обладает более высоким быстродействием и существенно более высокой ценой, чем магнитные носители, такие как магнитные диски или ленты. Большинство компьютеров, помимо первичной памяти, оснащено ***вторичной памятью*** (secondary storage) на базе магнитных дисков. Ее суммарный объем в типичной вычислительной системе как минимум на пару порядков превышает объем первичной памяти.

На рис. 18.2 показан типичный дисковый накопитель, состоящий из нескольких ***дисков*** (platters), вращающихся с постоянной скоростью на общем ***шпинделе*** (spindle). Поверхность каждого диска покрыта магнитным материалом. Каждый диск читается и записывается с помощью магнитной ***головки*** (head), расположенной на специальном ***рычаге*** (arm). Все рычаги с головками собраны в единый пакет, который позволяет перемещать головки вдоль радиуса по направлению к шпинделю или от него к краю дисков. Когда головки находятся в зафиксированном состоянии, поверхность, проходящая под ними при вращении дисков,

называется ***дорожкой*** (track). Наличие нескольких дисков в одном накопителе увеличивает его емкость, но не его производительность.

Хотя диски существенно дешевле оперативной памяти и имеют высокую емкость, из-за механически движущихся частей они гораздо, гораздо медленнее оперативной памяти.[1] Механическое движение головки относительно диска определяется двумя компонентами — перемещением головки по радиусу и вращением дисков. Когда были написаны эти строки, типичная скорость вращения дисков составляла 5400–15 000 об/мин (rpm). Обычно диски со скоростью 15 000 об/мин можно встретить в мощных серверах, скорость 7200 обычна для настольных систем, а 5400 — для переносных. Хотя скорость 7200 об/мин может показаться очень большой, один оборот требует примерно 8.33 мс, что более чем на 5 порядков превышает время обращения к оперативной памяти (которое обычно составляет примерно 50 нс). Другими словами, пока мы ждем оборота диска, чтобы считать необходимые нам данные, из оперативной памяти мы могли бы получить эти данные почти 100 000 раз! В среднем приходится ждать только половину оборота диска, но это практически ничего не меняет. Радиальное перемещение головок также требует времени. Одним словом, во время написания этих строк среднее время доступа к дисковой памяти для распространенных дисков составляло 8–11 мс.

Для того чтобы сократить время ожидания, связанное с механическим перемещением, при обращении к диску выполняется обращение сразу к нескольким элементам, хранящимся на диске. Информация разделяется на несколько ***страниц*** (pages) одинакового размера, которые хранятся последовательно одна за другой в пределах одной дорожки, и каждая операция чтения или записи работает с одной или несколькими страницами. Для типичного диска размер страницы может составлять 2^{11}–2^{14} байт. После того как головка позиционируется на нужную дорожку, а диск поворачивается так, что головка становится на начало интересующей нас страницы, операции чтения и записи выполняются очень быстро.

Зачастую обработка прочитанной информации длится меньше времени, чем ее поиск и чтение с диска. По этой причине в данной главе мы отдельно рассматриваем два компонента времени работы алгоритма:

- количество обращений к диску;
- время вычислений (процессорное время).

Количество обращений к диску измеряется в количествах страниц информации, которое должно быть считано с диска или записано на него. Заметим, что время обращения к диску не является постоянной величиной, поскольку зависит от расстояния между текущей дорожкой и дорожкой с интересующей нас информацией, а также от текущего угла поворота диска. Мы будем игнорировать это обстоятельство и в качестве первого приближения времени, необходимого для

---

[1]На момент написания книги на потребительский рынок стали активно выходить твердотельные накопители. Хотя они и быстрее дисковых, они отличаются более высокой стоимостью в пересчете на гигабайт и меньшей емкостью, чем традиционные магнитные диски.

обращения к диску, будем использовать просто количество считываемых или записываемых страниц.

В типичном приложении, использующем B-деревья, количество обрабатываемых данных так велико, что все они не могут одновременно разместиться в оперативной памяти. Алгоритмы работы с B-деревьями копируют в оперативную память с диска только некоторые выбранные страницы, необходимые для работы, и вновь записывают на диск те из них, которые были изменены в процессе работы. Алгоритмы работы с B-деревьями сконструированы таким образом, чтобы в любой момент времени обходиться только некоторым постоянным количеством страниц в основной памяти, так что ее объем не ограничивает размер B-деревьев, с которыми могут работать алгоритмы.

В нашем псевдокоде мы моделируем дисковые операции следующим образом. Пусть x — указатель на объект. Если объект находится в оперативной памяти компьютера, то мы обращаемся к его атрибутам обычным способом, например как к $x.key$. Если же объект, к которому мы обращаемся посредством x, находится на диске, то мы должны выполнить операцию DISK-READ(x) для чтения объекта x в оперативную память перед тем, как будем обращаться к его полям. (Считаем, что если объект x уже находится в оперативной памяти, то операция DISK-READ(x) не требует обращения к диску.) Аналогично для сохранения любых изменений в полях объекта x выполняется операция DISK-WRITE(x). Таким образом, типичный шаблон работы с объектом x имеет следующий вид.

```
x = a pointer to some object
DISK-READ(x)
Операции, обращающиеся к
    атрибутам x и/или изменяющие их
DISK-WRITE(x)  // Не выполняется, если никакие
               // атрибуты x не были изменены
Прочие операции, обращающиеся к атрибутам x,
    но не изменяющие их
```

Система в состоянии поддерживать в процессе работы в оперативной памяти только некоторое ограниченное количество страниц. Мы будем считать, что страницы, которые более не используются, удаляются из оперативной памяти системой; наши алгоритмы работы с B-деревьями не будут заниматься этим самостоятельно.

Поскольку в большинстве систем время выполнения алгоритма, работающего с B-деревьями, зависит, в первую очередь, от количества выполняемых операций с диском DISK-READ и DISK-WRITE, желательно минимизировать их количество и за один раз считывать и записывать как можно больше информации. Таким образом, размер узла B-дерева обычно соответствует дисковой странице. Количество потомков узла B-дерева, таким образом, ограничивается размером дисковой страницы.

Для больших B-деревьев, хранящихся на диске, степень ветвления обычно находится между 50 и 2000, в зависимости от размера ключа относительно размера страницы. Большáя степень ветвления резко снижает как высоту дерева,

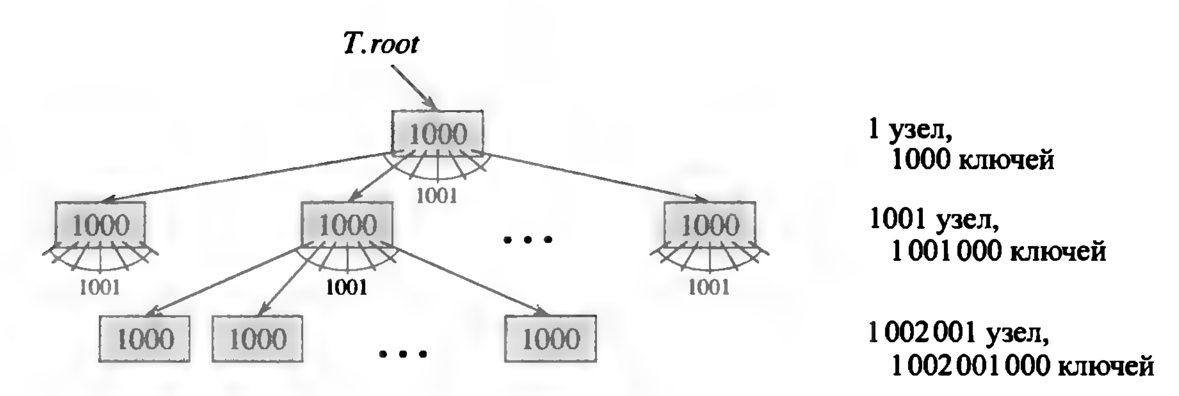


Рис. 18.3. В-дерево высотой 2, содержащее более миллиарда ключей. В каждом узле x показан атрибут $x.n$, количество ключей в x. Каждый внутренний узел и лист содержат 1000 ключей. Данное В-дерево имеет 1001 узел на глубине 1 и более миллиона листьев на глубине 2.

так и количество обращений к диску для поиска ключа. На рис. 18.3 показано В-дерево, высота которого равна 2, а степень ветвления — 1001; такое дерево может хранить более миллиарда ключей. При этом, поскольку корневой узел может храниться в оперативной памяти постоянно, для поиска ключа в этом дереве требуется максимум два обращения к диску.

18.1. Определение В-деревьев

Для простоты предположим, как и в случае бинарных деревьев поиска и красно-черных деревьев, что сопутствующая информация, связанная с ключом, хранится в узле вместе с ключом. На практике вместе с ключом может храниться только указатель на другую дисковую страницу, содержащую сопутствующую информацию для данного ключа. Псевдокод в данной главе неявно подразумевает, что при перемещении ключа от одного узла к другому вместе с ним перемещается и сопутствующая информация или указатель на нее. В распространенном варианте В-дерева, который называется ***B^+-деревом***, вся сопутствующая информация хранится в листьях, а во внутренних узлах хранятся только ключи и указатели на дочерние узлы. Таким образом, удается получить максимально возможную степень ветвления во внутренних узлах.

В-дерево T представляет собой корневое дерево (корень которого $T.root$), обладающее следующими свойствами.

1. Каждый узел x содержит следующие атрибуты:

   а) $x.n$, количество ключей, хранящихся в настоящий момент в узле x;

   б) собственно $x.n$ ключей, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, хранящихся в неубывающем порядке, так что $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$;

   в) булево значение $x.leaf$, равное TRUE, если x представляет собой лист, и FALSE, если x является внутренним узлом.

2. Кроме того, каждый внутренний узел x содержит $x.n+1$ указателей $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ на дочерние узлы. У листьев дочерних узлов нет, так что значения их атрибутов c_i не определены.

3. Ключи $x.key_i$ разделяют поддиапазоны ключей, хранящихся в поддеревьях: если k_i является произвольным ключом, хранящимся в поддереве с корнем $x.c_i$, то

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1} .$$

4. Все листья расположены на одной и той же глубине, которая равна высоте дерева h.

5. Имеются нижняя и верхняя границы количества ключей, которые могут содержаться в узле. Эти границы могут быть выражены с помощью одного фиксированного целого числа $t \geq 2$, называемого ***минимальной степенью*** (minimum degree) B-дерева:

   а) каждый узел, кроме корневого, должен содержать как минимум $t-1$ ключей. Каждый внутренний узел, не являющийся корневым, имеет, таким образом, как минимум t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ;

   б) каждый узел содержит не более $2t - 1$ ключей. Таким образом, внутренний узел имеет не более $2t$ дочерних узлов. Мы говорим, что узел ***заполнен*** (full), если он содержит ровно $2t - 1$ ключей[2].

Простейшее B-дерево получается при $t = 2$. При этом каждый внутренний узел может иметь 2, 3 или 4 дочерних узла, и мы получаем так называемое ***2-3-4-дерево*** (2-3-4 tree). Однако обычно на практике используются гораздо бóльшие значения t.

Высота B-дерева

Количество обращений к диску, необходимое для выполнения большинства операций с B-деревом, пропорционально его высоте. Проанализируем высоту B-дерева в наихудшем случае.

Теорема 18.1

Если $n \geq 1$, то для любого B-дерева T с n ключами, высотой h и минимальной степенью $t \geq 2$

$$h \leq \log_t \frac{n+1}{2} .$$

---

[2]Другой распространенный вариант B-дерева под названием ***B*-дерева*** требует, чтобы каждый внутренний узел был заполнен как минимум на две трети, а не наполовину, как в случае B-дерева.

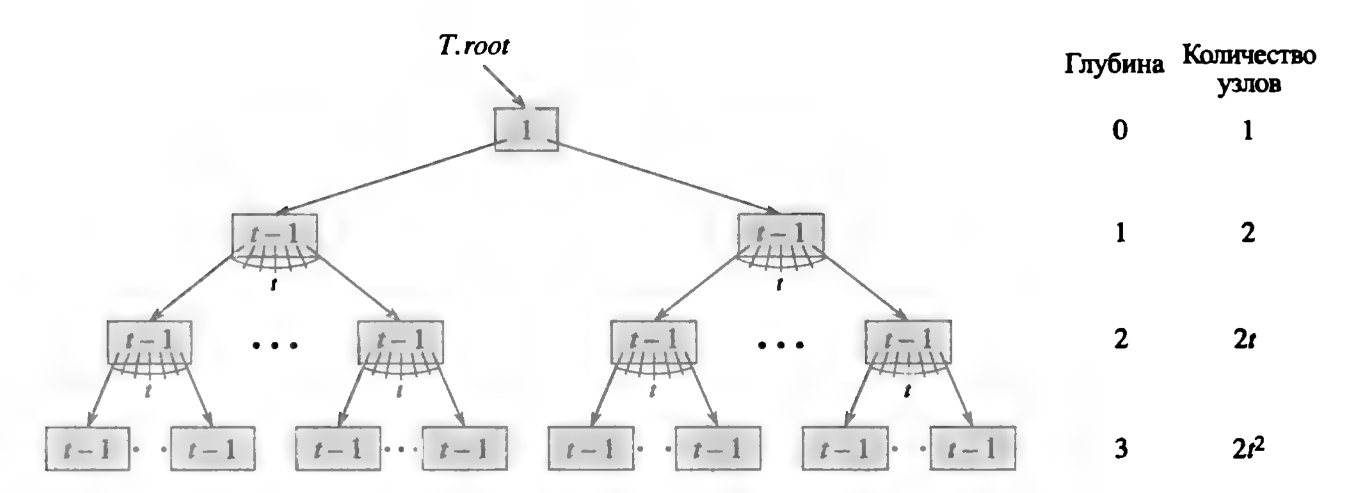


Рис. 18.4. B-дерево высотой 3, содержащее минимально возможное количество ключей. В каждом узле x показано значение $x.n$.

Доказательство. Корень B-дерева T содержит как минимум один ключ, а все остальные узлы — как минимум по $t-1$ ключей. Таким образом, у B-дерева T, глубина которого равна h, имеется как минимум 2 узла на глубине 1, как минимум $2t$ узлов на глубине 2, как минимум $2t^2$ узлов на глубине 3 и так до глубины h, на которой имеется как минимум $2t^{h-1}$ узлов (пример такого дерева, высота которого равна 3, показан на рис. 18.4). Следовательно, число ключей n удовлетворяет следующему неравенству:

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \; . \end{aligned}$$

Простейшее преобразование дает нам неравенство $t^h \leq (n+1)/2$. Теорема доказывается путем логарифмирования по основанию t обеих частей этого неравенства. ∎

Здесь мы видим преимущества B-деревьев над красно-черными деревьями. Хотя высота деревьев растет как $O(\lg n)$ в обоих случаях (вспомним, что t — константа), в случае B-деревьев основание логарифмов имеет гораздо большее значение. Таким образом, B-деревья требуют исследования примерно в $\lg t$ раз меньшего количества узлов по сравнению с красно-черными деревьями. Поскольку исследование узла дерева обычно требует обращения к диску, количество дисковых операций при работе с B-деревьями оказывается существенно сниженным.

Упражнения

18.1.1

Почему минимальная степень B-дерева не может быть $t = 1$?

18.1.2

Для каких значений t дерево на рис. 18.1 является корректным B-деревом?

18.1.3

Изобразите все корректные B-деревья с минимальной степенью 2, представляющие множество $\{1, 2, 3, 4, 5\}$.

18.1.4

Чему равно максимальное количество ключей, которое может храниться в B-дереве высотой h, выраженное в виде функции от минимальной степени дерева t?

18.1.5

Опишите структуру данных, которая получится, если в красно-черном дереве каждый черный узел поглотит красные дочерние узлы, а их потомки станут дочерними узлами этого черного узла.

18.2. Основные операции с B-деревьями

В этом разделе мы более подробно рассмотрим операции B-TREE-SEARCH, B-TREE-CREATE и B-TREE-INSERT, приняв при этом следующие соглашения.

- Корень B-дерева всегда находится в оперативной памяти, так что операция DISK-READ для чтения корневого узла не нужна; однако при изменении корневого узла требуется выполнение операции DISK-WRITE, сохраняющей внесенные изменения на диске.
- Все узлы, передаваемые в качестве параметров, уже считаны с диска операцией DISK-READ.

Все представленные здесь процедуры выполняются за один нисходящий проход по дереву.

Поиск в B-дереве

Поиск в B-дереве очень похож на поиск в бинарном дереве поиска, но с тем отличием, что если в бинарном дереве поиска мы выбирали один из двух путей, то здесь предстоит сделать выбор из большего количества альтернатив, зависящего от того, сколько дочерних узлов имеется у текущего узла. Точнее, в каждом внутреннем узле x нам предстоит выбрать одну из $(x.n + 1)$ ветвей.

Операция B-TREE-SEARCH представляет собой естественное обобщение процедуры TREE-SEARCH, определенной для бинарных деревьев поиска. В качестве параметров процедура B-TREE-SEARCH получает указатель на корневой узел x поддерева и ключ k, который следует найти в этом поддереве. Таким образом, вызов верхнего уровня для поиска во всем дереве имеет вид B-TREE-SEARCH$(T.root, k)$. Если ключ k имеется в B-дереве, процедура B-TREE-

SEARCH вернет упорядоченную пару (y, i), состоящую из узла y и индекса i, такого, что $y.key_i = k$. В противном случае процедура вернет значение NIL.

B-TREE-SEARCH(x, k)

1 $i = 1$
2 **while** $i \leq x.n$ и $k > x.key_i$
3 $\quad i = i + 1$
4 **if** $i \leq x.n$ и $k == x.key_i$
5 $\quad$ **return** (x, i)
6 **elseif** $x.leaf$
7 $\quad$ **return** NIL
8 **else** DISK-READ$(x.c_i)$
9 $\quad$ **return** B-TREE-SEARCH$(x.c_i, k)$

В строках 1–3 выполняется линейный поиск наименьшего индекса i, такого, что $k \leq x.key_i$ (иначе i присваивается значение $x.n + 1$). В строках 4 и 5 проверяется, не найден ли ключ в текущем узле, и если найден, то выполняется его возврат. В противном случае в строках 6–9 процедура либо завершает свою работу неудачей (если x является листом), либо рекурсивно вызывается для поиска в соответствующем поддереве x (после выполнения чтения с диска необходимого дочернего узла, являющегося корнем исследуемого поддерева).

На рис. 18.1 показана работа процедуры B-TREE-SEARCH: светлым цветом выделены узлы, просмотренные в процессе поиска ключа R.

Процедура B-TREE-SEARCH, как и процедура TREE-SEARCH при выполнении поиска в бинарном дереве, проходит в процессе рекурсии узлы дерева от корня в нисходящем порядке. Количество дисковых страниц, к которым выполняется обращение процедурой B-TREE-SEARCH, равно $O(h) = O(\log_t n)$, где h — высота В-дерева, а n — количество содержащихся в нем узлов. Поскольку $x.n < 2t$, количество итераций цикла **while** в строках 2 и 3 в каждом узле равно $O(t)$, а общее время вычислений составляет $O(th) = O(t \log_t n)$.

Создание пустого В-дерева

Для построения В-дерева T сначала необходимо воспользоваться процедурой B-TREE-CREATE для создания пустого корневого узла, а затем внести в него новые ключи с помощью процедуры B-TREE-INSERT. В обеих этих процедурах используется вспомогательная процедура ALLOCATE-NODE, которая выделяет дисковую страницу для нового узла за время $O(1)$. Мы можем считать, что эта процедура не требует вызова DISK-READ, поскольку никакой полезной информации о новом узле на диске при этом не сохраняется.

B-TREE-CREATE(T)

$x = $ ALLOCATE-NODE()
$x.leaf = $ TRUE
$x.n = 0$
DISK-WRITE(x)
$T.root = x$

B-TREE-CREATE требует $O(1)$ дисковых операций, время ее выполнения также равно $O(1)$.

Вставка ключа в B-дерево

Вставка ключа в B-дерево существенно сложнее вставки в бинарное дерево поиска. Как и в случае бинарных деревьев поиска, мы ищем позицию листа, в который будет вставлен новый ключ. Однако при работе с B-деревом мы не можем просто создать лист и вставить в него ключ, поскольку такое дерево не будет являться корректным B-деревом. Вместо этого мы вставляем новый ключ в существующий лист. Поскольку вставить новый ключ в заполненный лист невозможно, мы вводим новую операцию — ***разбиение*** (splitting) заполненного (т.е. содержащего $2t - 1$ ключей) узла y на два, каждый из которых содержит по $t - 1$ ключей. ***Медиана***, или средний ключ (median key), $y.key_t$ при этом перемещается в родительский по отношению к y узел, где становится разделительной точкой для двух вновь образовавшихся поддеревьев. Однако если родительский узел заполнен, перед вставкой нового ключа его также следует разбить, и такой процесс разбиения может выполняться по восходящей до самого корня.

Как и в случае бинарного дерева поиска, в B-дереве мы вполне можем осуществить вставку за один нисходящий проход от корня к листу. Для этого не нужно выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. Вместо этого при проходе от корня к листьям в поисках позиции для нового ключа мы разбиваем все заполненные узлы, через которые проходим (включая сам лист). Тем самым гарантируется, что если потребуется разбить какой-то узел, то его родительский узел заполнен не будет.

Разбиение узла B-дерева

Процедура B-TREE-SPLIT-CHILD получает в качестве входного параметра *незаполненный* внутренний узел x (находящийся в оперативной памяти) и индекс i, такой, что узел $x.c_i$ (также находящийся в оперативной памяти) является заполненным дочерним узлом x. Процедура разбивает дочерний узел на два и соответствующим образом обновляет поля x, внося в него информацию о новом дочернем узле. Для разбиения заполненного корневого узла мы сначала делаем корень дочерним узлом нового пустого корневого узла, после чего можем использовать вызов B-TREE-SPLIT-CHILD. При этом высота дерева увеличивается на 1. Разбиение — единственный путь увеличения высоты B-дерева.

На рис. 18.5 показан этот процесс. Заполненный узел y разбивается медианой S, которая перемещается в родительский по отношению к y узел. Те ключи

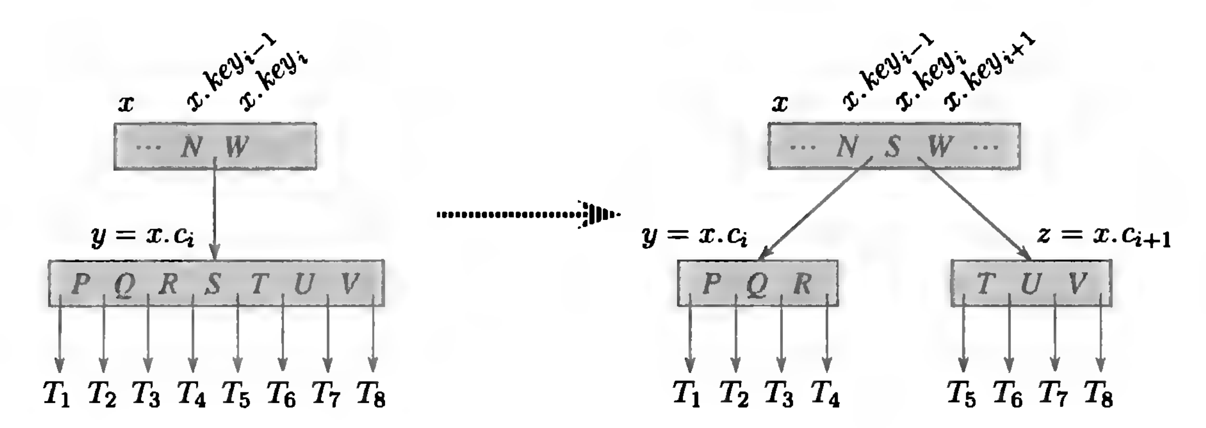


Рис. 18.5. Разбиение узла с $t = 4$. Узел $y = x.c_i$ разбивается на два узла, y и z, и средний ключ S узла y перемещается в родительский по отношению к y узел.

из y, которые больше медианы, помещаются в новый узел z, который становится новым дочерним узлом x.

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

Процедура B-TREE-SPLIT-CHILD использует простой способ “вырезать и вставить”. Здесь x является разбиваемым узлом, а y представляет собой i-й дочерний узел узла x (устанавливается в строке 2). Изначально узел y имеет $2t$ дочерних узла (содержит $2t - 1$ ключей); после разбиения количество его дочерних узлов уменьшится до t ($t - 1$ ключей). Узел z получает t бóльших дочерних узлов ($t - 1$ ключей) y и становится новым дочерним узлом x, располагаясь непосредственно

после y в таблице дочерних узлов x. Медиана y перемещается в узел x и разделяет в нем y и z.

В строках 1–9 создается узел z и в него переносятся бóльшие $t-1$ ключей и соответствующие t дочерних узлов y. В строке 10 обновляется количество ключей в y. И наконец строки 11–17 вставляют z в качестве дочернего узла x, перенося медиану из y в x для разделения y и z, и обновляют количество ключей в x. В строках 18–20 выполняется запись на диск всех модифицированных страниц. Процессорное время работы процедуры равно $\Theta(t)$ из-за циклов в строках 5 и 6, а также 8 и 9 (прочие циклы выполняют $O(t)$ итераций). Процедура выполняет $O(1)$ дисковых операций.

Вставка ключа в В-дерево за один проход

Вставка ключа k в В-дерево T высотой h выполняется за один нисходящий проход по дереву, требующий $O(h)$ обращений к диску. Необходимое процессорное время составляет $O(th) = O(t \log_t n)$. Процедура B-TREE-INSERT использует процедуру B-TREE-SPLIT-CHILD для гарантии того, что рекурсия никогда не столкнется с заполненным узлом.

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t - 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

Строки 3–9 обрабатывают случай, когда заполнен корень дерева r: при этом корень разбивается и новый узел s (у которого два дочерних узла) становится новым корнем В-дерева. Высота В-дерева может увеличиваться только при разбиении корня. На рис. 18.6 проиллюстрирован такой процесс разбиения корневого узла. В отличие от бинарных деревьев поиска, высота В-деревьев увеличивается "сверху", а не "снизу". Завершается процедура вызовом другой процедуры — B-TREE-INSERT-NONFULL, — которая выполняет вставку ключа k в дерево с незаполненным корневым узлом. Данная процедура при необходимости рекурсивно спускается вниз по дереву, причем каждый узел, в который она входит, является незаполненным, что обеспечивается (при необходимости) вызовом процедуры B-TREE-SPLIT-CHILD.

Вспомогательная рекурсивная процедура B-TREE-INSERT-NONFULL вставляет ключ k в узел x, который при вызове процедуры должен быть незаполненным. Операции B-TREE-INSERT и B-TREE-INSERT-NONFULL гарантируют, что это условие незаполненности будет выполнено.

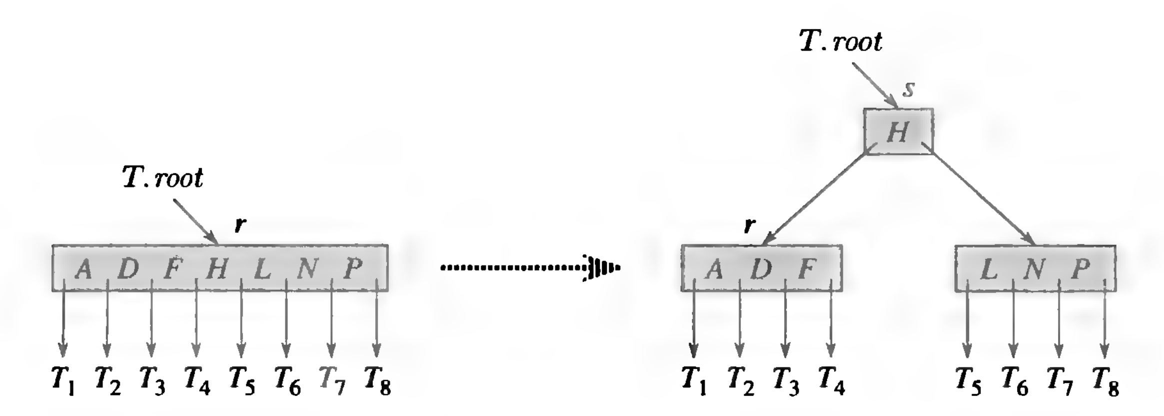


Рис. 18.6. Разбиение корня при $t = 4$. Корневой узел r разбивается на два, и создается новый корневой узел s. Новый корень содержит медианный ключ r, а две половины r становятся его дочерними узлами. Высота В-дерева при разбиении корня увеличивается на единицу.

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 и k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i − 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 и k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

Процедура B-TREE-INSERT-NONFULL работает следующим образом. Строки 3–8 обрабатывают случай, когда x является листом; при этом ключ k просто вставляется в данный лист. Если же x не является листом, то необходимо вставить k в подходящий лист в поддереве, корнем которого является внутренний узел x. В этом случае строки 9–11 определяют дочерний узел x, в который спустится рекурсия. В строке 13 проверяется, не заполнен ли этот дочерний узел, и если он заполнен, то в строке 14 вызывается процедура B-TREE-SPLIT-CHILD, которая разбивает его на два незаполненных узла, а строки 15 и 16 определяют, в какой из двух полученных в результате разбиения узлов должна спуститься рекурсия. (Обратите внимание, что в строке 16 после увеличения i операция чтения с диска DISK-READ$(x.c_i)$ не нужна, поскольку процедура рекурсивно спускается в узел, только что созданный процедурой B-TREE-SPLIT-CHILD.) Строки 13–16 гарантируют, что процедура никогда не столкнется с заполненным узлом. Затем строка 17

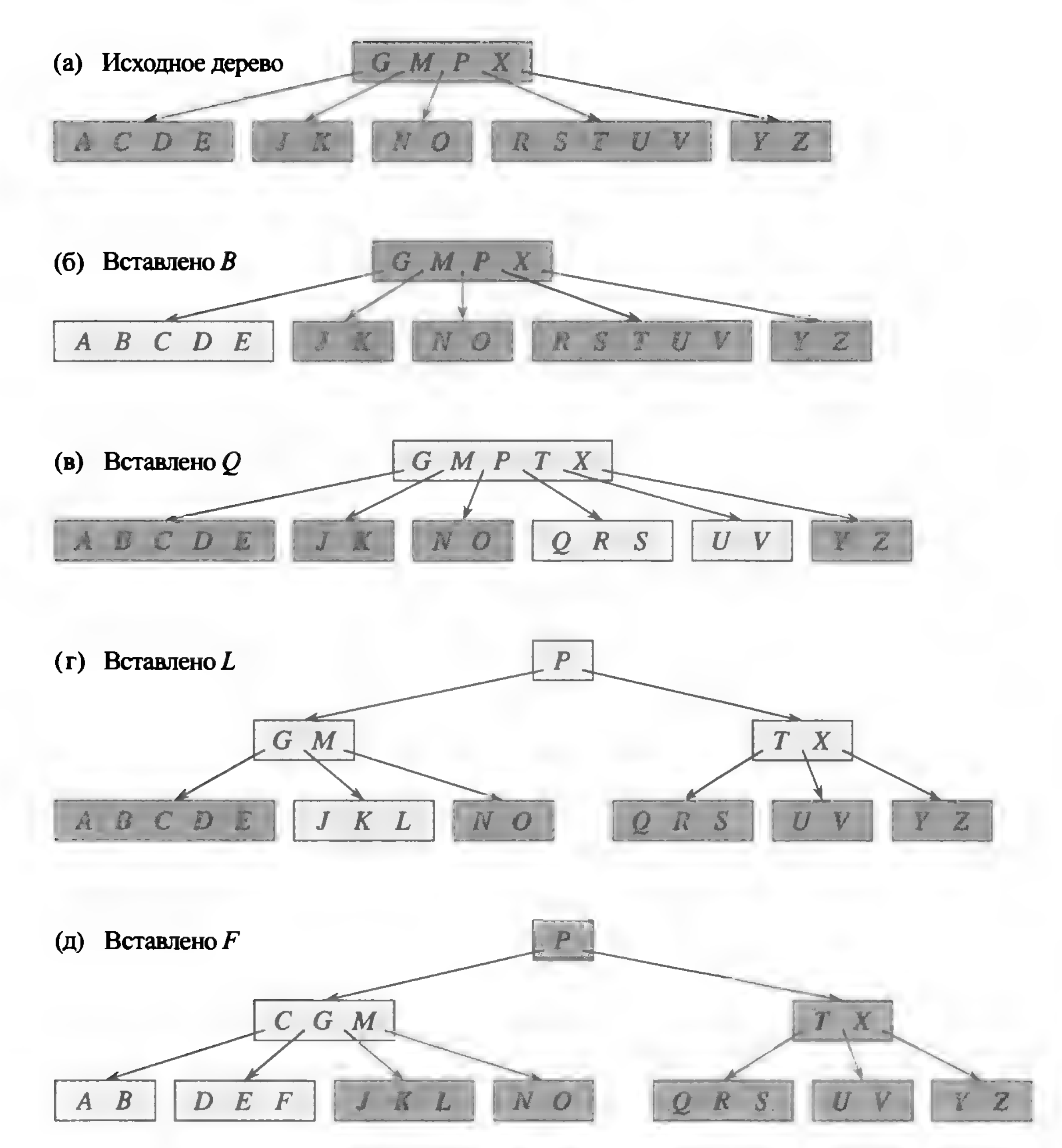


Рис. 18.7. Вставка ключей в B-дерево с минимальной степенью $t = 3$, так что узел может содержать не более пяти ключей. Узлы, модифицированные в процессе вставки, выделены светлой штриховкой. **(а)** Исходное дерево для приведенного примера. **(б)** Результат вставки B в исходное дерево. Это простая вставка в лист. **(в)** Результат вставки Q в предыдущее дерево. Узел $RSTUV$ разбивается на два узла, содержащих RS и UV, ключ T переносится вверх в корень, а Q вставляется в левую половину (узел RS). **(г)** Результат вставки L в предыдущее дерево. Корень, будучи заполненным, разбивается, и высота B-дерева увеличивается на единицу. Затем L вставляется в лист, содержащий JK. **(д)** Результат вставки F в предыдущее дерево. Узел $ABCDE$ разбивается перед вставкой F в правую половину (узел DE).

рекурсивно вызывает процедуру B-TREE-INSERT-NONFULL для вставки k в соответствующее поддерево. На рис. 18.7 проиллюстрированы различные ситуации, возникающие при вставке в B-дерево.

Количество обращений к диску, выполняемых процедурой B-TREE-INSERT для B-дерева высотой h, составляет $O(h)$, поскольку между вызовами B-TREE-INSERT-NONFULL выполняется только $O(1)$ операций DISK-READ и DISK-WRITE. Необходимое процессорное время равно $O(th) = O(t \log_t n)$. Поскольку в про-

цедуре B-TREE-INSERT-NONFULL использована оконечная рекурсия, ее можно реализовать итеративно с помощью цикла **while**, тем самым демонстрируя, что количество страниц, которые должны находиться в оперативной памяти, в любой момент времени равно $O(1)$.

Упражнения

18.2.1

Покажите результат вставки ключей

$$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$

в указанном порядке в изначально пустое В-дерево с минимальной степенью 2. Изобразите только конфигурации дерева непосредственно перед выполнением разбиения и окончательный вид дерева.

18.2.2

Поясните, при каких условиях могут выполняться излишние операции DISK-READ и DISK-WRITE в процессе работы процедуры B-TREE-INSERT (если таковые могут иметь место). Под излишней операцией чтения подразумевается чтение страницы, уже находящейся в оперативной памяти, а под излишней записью — запись на диск информации, идентичной уже имеющейся там.

18.2.3

Как найти минимальный ключ в В-дереве? Как найти элемент В-дерева, предшествующий данному ключу?

18.2.4 ★

Предположим, что ключи $\{1, 2, \ldots, n\}$ вставлены в пустое В-дерево с минимальной степенью 2. Сколько узлов будет иметь полученное в результате В-дерево?

18.2.5

Поскольку листья не содержат указателей на дочерние узлы, в них можно разместить больше ключей, чем во внутренних узлах при том же размере дисковой страницы. Покажите, как следует изменить процедуры создания В-дерева и вставки в него для работы с таким видоизмененным В-деревом.

18.2.6

Предположим, что линейный поиск в узле в процедуре B-TREE-SEARCH заменен бинарным поиском. Покажите, что при этом процессорное время, необходимое для выполнения процедуры B-TREE-SEARCH, равно $O(\lg n)$ (и не зависит от t).

18.2.7

Предположим, что дисковое аппаратное обеспечение позволяет произвольно выбирать размер дисковой страницы, но время, необходимое для чтения дисковой страницы, равно $a + bt$, где a и b — определенные константы, а t — минимальная степень В-дерева, использующего страницу данного размера. Как следует вы-

брать t, чтобы (приближенно) минимизировать время поиска в В-дереве? Оцените оптимальное значение t для $a = 5$ мс и $b = 10$ мкс.

18.3. Удаление ключа из В-дерева

Удаление ключа из В-дерева, хотя и аналогично вставке, представляет собой более сложную задачу. Это связано с тем, что ключ может быть удален из любого узла, а не только из листа, а удаление из внутреннего узла требует определенной перестройки дочерних узлов. Как и в случае вставки, мы должны обеспечить сохранение свойств В-дерева при выполнении операции удаления. Аналогично тому, как мы обеспечивали отсутствие переполнения при вставке нового ключа, нам предстоит обеспечивать условие, чтобы узел не становился слишком мало заполненным в процессе удаления ключа (за исключением корневого узла, который может иметь менее $t - 1$ ключей). Так же, как алгоритму вставки может потребоваться возврат, если узел на пути вставки ключа заполнен, так и алгоритму удаления может потребоваться возврат, если узел (не корневой) на пути вставки ключа имеет минимально допустимое количество ключей.

Итак, пусть процедура B-TREE-DELETE должна удалить ключ k из поддерева, корнем которого является узел x. Эта процедура разработана таким образом, что при ее рекурсивном вызове для узла x гарантировано наличие в этом узле по меньшей мере t ключей. Заметим, что это условие требует наличия в узле количества ключей, на один больше минимально требуемого свойствами В-дерева, так что иногда ключ может быть перемещен в дочерний узел перед тем, как рекурсия опустится в этот дочерний узел. Такое ужесточение свойства В-дерева (наличие "запасного" ключа) дает нам возможность выполнить удаление ключа за один нисходящий проход по дереву (с единственным исключением, которое будет пояснено позже). Следует также учесть, что если корень дерева x становится внутренним узлом, не содержащим ни одного ключа (такая ситуация может возникнуть в рассматриваемых ниже случаях 2, (в) и 3, (б) на с. 537), то узел x удаляется, а его единственный дочерний узел $x.c_1$ становится новым корнем дерева (при этом уменьшается высота В-дерева и сохраняется его свойство, требующее, чтобы корневой узел непустого дерева содержал как минимум один ключ).

Вместо того чтобы представить вам полный псевдокод процедуры удаления узла из В-дерева, мы просто набросаем последовательность выполняемых действий. На рис. 18.8 показаны возникающие при удалении ключа из В-дерева ситуации.

1. Если ключ k находится в узле x и x является листом, удаляем ключ k из x.
2. Если ключ k находится в узле x и x является внутренним узлом, делаем следующее.

   а) Если дочерний по отношению к x узел y, предшествующий ключу k в узле x, содержит не менее t ключей, то находим k' — предшественника k в поддереве, корнем которого является y. Рекурсивно удаляем k' и заменяем k

в x ключом k'. (Поиск ключа k' и его удаление можно выполнить за один нисходящий проход.)

б) Если y имеет менее t ключей, то симметрично обращаемся к дочернему по отношению к x узлу z, который следует за ключом k в узле x. Если z содержит не менее t ключей, то находим k' — следующий за k ключ в поддереве, корнем которого является z. Рекурсивно удаляем k' и заменяем k в x ключом k'. (Поиск ключа k' и его удаление можно выполнить за один нисходящий проход.)

в) В противном случае, если и y, и z содержат по $t-1$ ключей, вносим k и все ключи z в y (при этом из x удаляются и k, и указатель на z, а узел y после этого содержит $2t-1$ ключей), а затем освобождаем z и рекурсивно удаляем k из y.

3. Если ключ k отсутствует во внутреннем узле x, находим корень $x.c_i$ поддерева, которое должно содержать k (если таковой ключ имеется в данном В-дереве). Если $x.c_i$ содержит только $t-1$ ключей, выполняем п. 3, (а) или (б) для того, чтобы гарантировать, что далее мы переходим в узел, содержащий как минимум t ключей. Затем мы рекурсивно удаляем k из соответствующего дочернего по отношению к x узла.

а) Если $x.c_i$ имеет только $t-1$ ключей, но при этом один из его непосредственных соседей (под которым мы понимаем дочерний по отношению к x узел, отделенный от рассматриваемого ровно одним ключом-разделителем) содержит как минимум t ключей, передадим в $x.c_i$ ключ-разделитель между данным узлом и его непосредственным соседом из x, на его место поместим крайний ключ из соседнего узла и перенесем соответствующий указатель из соседнего узла в $x.c_i$.

б) Если и $x.c_i$, и оба его непосредственных соседа содержат по $t-1$ ключей, объединим $x.c_i$ с одним из его соседей (при этом бывший ключ-разделитель из x будет перенесен вниз и станет медианой нового узла).

Поскольку большинство ключей в В-дереве находится в листьях, можно ожидать, что на практике чаще всего удаления будут выполняться из листьев. Процедура B-TREE-DELETE в этом случае выполняется за один нисходящий проход по дереву, без возвратов. Однако при удалении ключа из внутреннего узла процедуре может потребоваться возврат к узлу, ключ из которого был удален, чтобы заменить ключ его предшественником или преемником (случаи 2, (а) и (б)).

Хотя описание процедуры выглядит достаточно запутанным, она требует всего лишь $O(h)$ дисковых операций для В-дерева высотой h, поскольку между рекурсивными вызовами процедуры выполняется только $O(1)$ вызовов процедур DISK-READ или DISK-WRITE. Необходимое процессорное время составляет $O(th) = O(t \log_t n)$.

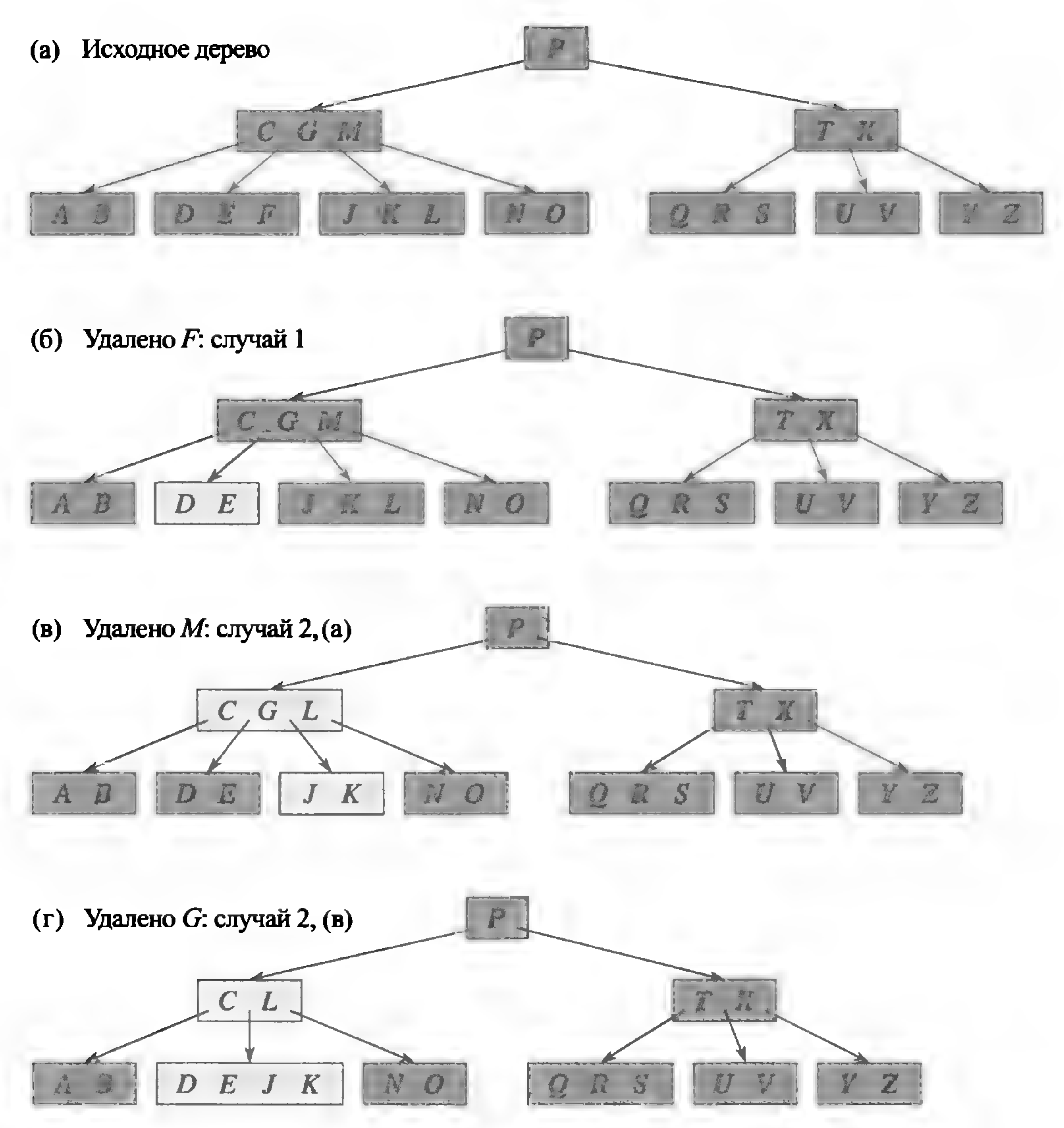


Рис. 18.8. Удаление ключей из В-дерева. Минимальная степень данного В-дерева $t = 3$, так что узел (отличный от корня) не может иметь меньше двух ключей. Модифицируемые узлы выделены светлой штриховкой. **(а)** В-дерево, приведенное на рис. 18.7, (д). **(б)** Удаление F. Реализуется случай 1: простое удаление из листа. **(в)** Удаление M. Реализуется случай 2, (а): предшественник M (L) перемещается вверх и занимает место M. **(г)** Удаление G. Реализуется случай 2, (в): G переносится вниз в узел $DEGJK$, а затем выполняется удаление G из листа (случай 1).

Упражнения

18.3.1

Изобразите результат удаления ключей C, P и V (в указанном порядке) из В-дерева, приведенного на рис. 18.8, (е).

18.3.2

Напишите псевдокод процедуры B-TREE-DELETE.

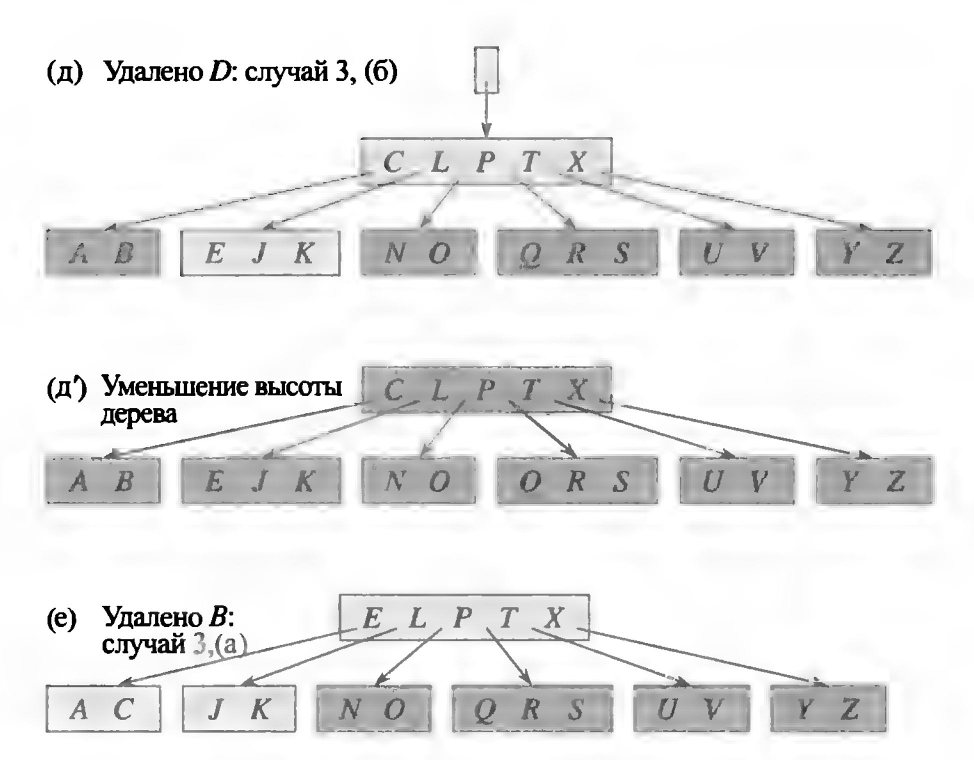


Рис. 18.8 (продолжение). **(д)** Удаление D. Реализуется случай 3, (б): рекурсия не может опуститься в узел CL, поскольку в нем только два узла, так что мы опускаем P и сливаем его с CL и TX с образованием $CLPTX$; затем мы удаляем D из листа (случай 1). **(д′)** После (д) удаляется корень, и высота дерева уменьшается на единицу. **(е)** Удаление B. Реализуется случай 3, (а): C перемещается для заполнения позиции B, а E перемещается для заполнения позиции C.

Задачи

18.1. Стеки во вторичной памяти

Рассмотрим реализацию стека в компьютере с небольшой оперативной памятью, но относительно большим дисковым пространством. Поддерживаемые стеком операции PUSH и POP должны работать со значениями, представляющими отдельные машинные слова. Стек должен иметь возможность роста до размеров, существенно превышающих размер оперативной памяти, так что его бóльшая часть должна храниться на диске.

Простая, но неэффективная реализация стека хранит его весь на диске. В памяти поддерживается только указатель стека, который представляет собой дисковый адрес элемента на вершине стека. Если указатель имеет значение p, элемент на вершине стека является $(p \bmod m)$-м словом на странице $\lfloor p/m \rfloor$ на диске, где m — количество слов в одной странице.

Для реализации операции PUSH мы увеличиваем указатель стека, считываем в оперативную память с диска соответствующую страницу, копируем вносимый в стек элемент в соответствующее место на странице и записываем страницу обратно на диск. Реализация операции POP аналогична реализации операции PUSH. Мы уменьшаем указатель стека, считываем в оперативную память с диска соответствующую страницу и возвращаем элемент на вершине стека. Нет необходимости записывать страницу на диск, поскольку она не была модифицирована.

Поскольку дисковые операции достаточно дорогие в смысле времени их выполнения, мы учитываем при любой реализации стека две стоимости: общее количество обращений к диску и необходимое процессорное время. Будем считать, что дисковые операции со страницей размером m слов требуют одного обращения к диску и процессорного времени $\Theta(m)$.

а. Чему в наихудшем случае равно асимптотическое количество обращений к диску при такой простейшей реализации стека? Чему равно требуемое процессорное время для n стековых операций? (Ваш ответ на этот и последующие вопросы данной задачи должен выражаться через m и n.)

Теперь рассмотрим реализацию стека, при которой одна страница стека находится в оперативной памяти (кроме того, небольшое количество оперативной памяти используется для отслеживания того, какая именно страница находится в оперативной памяти в настоящее время). Само собой разумеется, что мы можем выполнять стековые операции только в том случае, если соответствующая дисковая страница находится в оперативной памяти. При необходимости страница, находящаяся в данный момент в оперативной памяти, может быть записана на диск, а в память считана новая дисковая страница. Если нужная нам страница уже находится в оперативной памяти, то выполнение дисковых операций не требуется.

б. Чему в наихудшем случае равно количество необходимых обращений к диску для n операций PUSH? Чему равно соответствующее процессорное время?

в. Чему в наихудшем случае равно количество необходимых обращений к диску для n стековых операций? Чему равно соответствующее процессорное время?

Предположим, что стек реализован таким образом, что в оперативной памяти постоянно находятся две страницы (в дополнение к небольшому количеству оперативной памяти, которое используется для отслеживания того, какие именно страницы находятся в оперативной памяти в настоящее время).

г. Как следует управлять страницами стека, чтобы амортизированное количество обращений к диску для любой стековой операции составляло $O(1/m)$, а амортизированное процессорное время — $O(1)$?

18.2. Объединение и разделение 2-3-4-деревьев

Операция ***объединения*** (join) получает два динамических массива, S' и S'', и элемент x, такой, что для любых $x' \in S'$ и $x'' \in S''$ выполняется $x'.key < x.key < x''.key$. Операция объединения возвращает множество $S = S' \cup \{x\} \cup S''$. Операция ***разделения*** (split) представляет собой обращение операции объединения: для данного динамического множества S и элемента $x \in S$ она создает множество S', содержащее все элементы из $S - \{x\}$, ключи которых меньше $x.key$, и множество S'', состоящее из всех элементов $S - \{x\}$, ключи которых больше $x.key$. В данной задаче мы рассматриваем реализацию этих операций над 2-3-4-деревьями. Для удобства полагаем, что элементы представляют собой ключи и что все ключи различны.

а. Покажите, каким образом для каждого узла x поддерживать хранение в каждом узле 2-3-4-дерева в виде атрибута $x.height$ информации о высоте поддерева, корнем которого является узел x, чтобы это не повлияло на асимптотическое время выполнения операций поиска, вставки и удаления.

б. Покажите, как реализовать операцию объединения. Для данных 2-3-4-деревьев T' и T'' и ключа k объединение должно выполняться за время $O(1+|h'-h''|)$, где h' и h'' — значения высоты деревьев T' и T'' соответственно.

в. Рассмотрим простой путь p от корня 2-3-4-дерева T к заданному ключу k, множество S' ключей T, которые меньше k, и множество S'' ключей T, которые больше k. Покажите, что p разбивает S' на множество деревьев $\{T'_0, T'_1, \ldots, T'_m\}$ и множество ключей $\{k'_1, k'_2, \ldots, k'_m\}$, где для $i = 1, 2, \ldots, m$ мы имеем $y < k'_i < z$ для любых ключей $y \in T'_{i-1}$ и $z \in T'_i$. Как связаны высоты T'_{i-1} и T'_i? На какие множества деревьев и ключей p разбивает S''?

г. Покажите, как реализовать операцию разделения 2-3-4-дерева T. Воспользуйтесь операцией объединения для того, чтобы собрать ключи из S' в единое 2-3-4-дерево T', а ключи из S'' — в единое 2-3-4-дерево T''. Время работы операции разделения должно составлять $O(\lg n)$, где n — количество ключей в T. (*Указание:* при сложении стоимости операций объединения происходит сокращение членов телескопической суммы.)

Заключительные замечания

Сбалансированные деревья и В-деревья достаточно подробно рассмотрены в книгах Кнута (Knuth) [210][3], Ахо (Aho), Хопкрофта (Hopcroft), Ульмана (Ullman) [5] и Седжвика (Sedgewick) [304]. Подробный обзор В-деревьев дан Комером (Comer) [73]. Гибас (Guibas) и Седжвик [154] рассмотрели взаимосвязи между различными видами сбалансированных деревьев, включая красно-черные деревья и 2-3-4-деревья.

В 1970 году Хопкрофт ввел понятие 2-3-деревьев, которые были предшественниками В- и 2-3-4-деревьев и в которых каждый внутренний узел имел 2 или 3 дочерних узла. В-деревья предложены Баером (Bayer) и Мак-Крейтом (McCreight) в 1972 году [34] (выбор названия для своей разработки они не пояснили).

Бендер (Bender), Демейн (Demaine) и Фарах-Колтон (Farach-Colton) [39] изучили производительность В-деревьев при наличии иерархической памяти. Алгоритмы с ***игнорированием кеширования*** (cache-oblivious) эффективно работают без явного знания о размерах обмена данными между различными видами памяти.

[3]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 19. Фибоначчиевы пирамиды

Пирамиды Фибоначчи служат двум целям. Во-первых, они поддерживают набор операций, составляющих то, что известно под названием "объединяемые, или сливаемые, пирамиды" (mergeable heap). Во-вторых, некоторые операции в фибоначчиевых пирамидах выполняются за константное амортизированное время, что делает эту структуру данных хорошо подходящей для приложений, в которых часто применяются данные операции.

Объединяемые пирамиды

Объединяемой пирамидой является любая структура данных, поддерживающая следующие операции, в которых каждый элемент имеет ключ *key*.

MAKE-HEAP() создает и возвращает новую пирамиду, не содержащую элементов.

INSERT(H, x) вставляет в пирамиду H элемент x, ключ *key* которого к этому моменту заполнен.

MINIMUM(H) возвращает указатель на элемент пирамиды H, ключ которого минимален.

EXTRACT-MIN(H) удаляет из пирамиды H элемент, ключ которого минимален, и возвращает указатель на этот элемент.

UNION(H_1, H_2) создает и возвращает новую пирамиду, которая содержит все элементы пирамид H_1 и H_2. Пирамиды H_1 и H_2 этой операцией "уничтожаются".

В дополнение к перечисленным выше операциям с объединяемыми пирамидами пирамиды Фибоначчи поддерживают еще две операции.

DECREASE-KEY(H, x, k) назначает элементу x в пирамиде H новое значение ключа k, которое не больше его текущего значения.[1]

DELETE(H, x) удаляет элемент x из пирамиды H.

---

[1]Как упоминалось во введении к части V, по умолчанию рассматриваемые в книге объединяемые пирамиды являются объединяемыми неубывающими пирамидами, так что к ним применимы операции MINIMUM, EXTRACT-MIN и DECREASE-KEY. Можно точно так же определить объединяемые невозрастающие пирамиды с операциями MAXIMUM, EXTRACT-MAX и INCREASE-KEY.

Процедура	Бинарная пирамида (наихудший случай)	Фибоначчиева пирамида (амортизированная)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Рис. 19.1. Время выполнения операций в двух реализациях объединяемых пирамид. В момент выполнения операции количество элементов пирамиды (или пирамид) равно n.

Как показано в таблице на рис. 19.1, если операция UNION не требуется, то вполне можно воспользоваться обычной бинарной пирамидой, которая, например, применялась в пирамидальной сортировке (см. главу 6). Прочие, отличные от UNION, операции в бинарной пирамиде выполняются в наихудшем случае за время $O(\lg n)$. Однако при необходимости операции UNION производительности бинарной пирамиды недостаточно. В случае конкатенации массивов, хранящих объединяемые бинарные пирамиды, с последующим выполнением процедуры BUILD-MIN-HEAP (см. раздел 6.3) процедура UNION в наихудшем случае выполняется за время $\Theta(n)$.

С другой стороны, фибоначчиевы пирамиды имеют лучшие, чем у бинарных пирамид, асимптотические времена работы операций INSERT, UNION и DECREASE-KEY и такие же времена работы прочих операций, как и у бинарных пирамид. Заметим, однако, что времена работы для фибоначчиевых пирамид на рис. 19.1 представляют собой амортизированные значения, а не границы для наихудшего случая. Операция UNION в фибоначчиевой пирамиде требует для выполнения константного амортизированного времени, что значительно лучше, чем линейное время в наихудшем случае, требующееся в случае бинарной пирамиды (конечно, в предположении достаточности амортизированного времени работы).

Фибоначчиевы пирамиды в теории и на практике

С теоретической точки зрения фибоначчиевы пирамиды особенно полезны в случае, когда количество операций EXTRACT-MIN и DELETE относительно мало́ по сравнению с количеством других операций. Такая ситуация возникает во многих приложениях. Например, некоторые алгоритмы в задачах о графах могут вызывать процедуру DECREASE-KEY по одному разу для каждого ребра. В плотных графах с большим количеством ребер амортизированное время выполнения DECREASE-KEY, равное $\Theta(1)$, представляет собой существенный выигрыш по сравнению со временем $\Theta(\lg n)$ в наихудшем случае в биномиальных пирамидах. Быстрые алгоритмы для таких задач, как поиск минимального остовного дерева

(глава 23) или поиск кратчайших путей из одной вершины (глава 24), преимущественно опираются на фибоначчиевы пирамиды.

Однако с практической точки зрения программная сложность реализации и высокие значения постоянных множителей в формулах времени работы существенно снижают эффективность применения фибоначчиевых пирамид, делая их для большинства приложений менее привлекательными, чем обычные бинарные (или k-арные) пирамиды. Таким образом, интерес к фибоначчиевым пирамидам, в первую очередь, сугубо теоретический. Широкое практическое использование могла бы получить более простая структура данных, но обладающая тем же амортизированным временем работы, что и фибоначчиевы пирамиды.

И бинарные, и фибоначчиевы пирамиды недостаточно эффективно выполняют операцию поиска SEARCH; поиск элемента с заданным ключом может занять некоторое время. Именно поэтому такие операции, как DECREASE-KEY или DELETE, которые обращаются к конкретному элементу, требуют в качестве входных данных указатель на этот элемент. Как и в нашем рассмотрении очередей с приоритетами в разделе 6.5, при использовании в приложении сливаемых пирамид зачастую в каждом элементе пирамиды хранится идентификатор соответствующего объекта приложения, а идентификатор элемента пирамиды — в объекте приложения. Точная природа этих идентификаторов зависит от приложения и его реализации.

Подобно ряду других рассмотренных структур данных, фибоначчиевы пирамиды основаны на корневых деревьях. Мы представляем каждый элемент узлом в дереве, и каждый узел имеет атрибут key. В оставшейся части данной главы мы вместо термина "элемент" будем использовать термин "узел". Мы также проигнорируем вопросы выделения памяти для узла перед его вставкой и освобождения памяти после удаления, считая, что ответственность за эти действия возложена на код, вызывающий процедуры пирамиды.

В разделе 19.1 определяются фибоначчиевы пирамиды, рассматриваются их представление и функция потенциала, используемая в процессе выполнения амортизационного анализа. В разделе 19.2 показано, как реализовать операции объединяемых пирамид, получив при этом амортизированные времена работы, показанные на рис. 19.1. Две оставшиеся операции, DECREASE-KEY и DELETE, рассматриваются в разделе 19.3. Наконец, в разделе 19.4, завершается ключевая часть анализа, а также поясняется происхождение названия рассматриваемой структуры данных.

19.1. Структура фибоначчиевых пирамид

Фибоначчиева пирамида (Fibonacci heap) представляет собой набор корневых деревьев, упорядоченных в соответствии со ***свойством неубывающей пирамиды***, т.е. каждое дерево подчиняется этому свойству: ключ узла не меньше ключа его родителя. На рис. 19.2, (а) показан пример фибоначчиевой пирамиды.

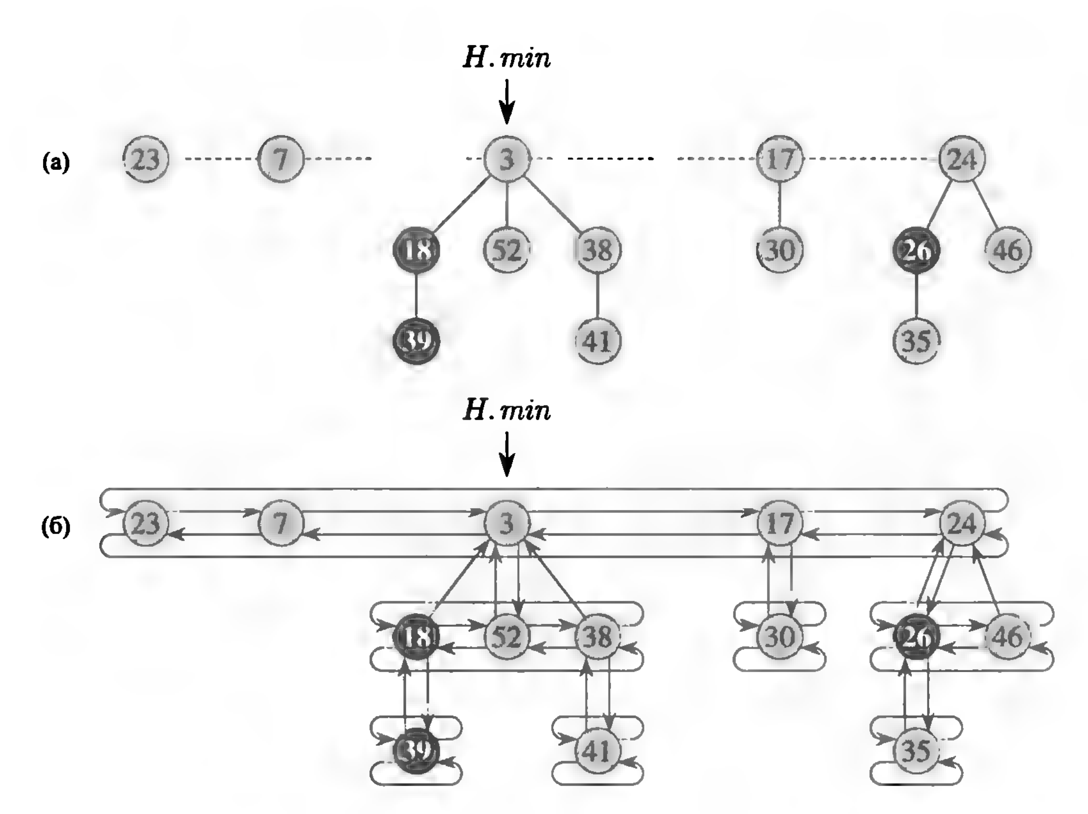


Рис. 19.2. **(а)** Фибоначчиева пирамида, состоящая из пяти деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды, и четырнадцати узлов. Пунктирная линия указывает корневой список. Минимальным узлом пирамиды является узел, содержащий ключ 3. Помеченные узлы показаны черным цветом. Потенциал данной конкретной фибоначчиевой пирамиды равен $5 + 2 \cdot 3 = 11$. **(б)** Более полное представление, показывающее указатели p (стрелки вверх), *child* (стрелки вниз) и *left* и *right* (горизонтальные стрелки). На прочих рисунках главы эти детали опущены, поскольку вся указанная информация может быть восстановлена из схемы наподобие приведенной в части (а).

Как показано на рис. 19.2,(б), каждый узел x содержит указатель $x.p$ на родительский узел и указатель $x.child$ на некоторый из дочерних узлов. Дочерние узлы узла x циклически соединены в двусвязный список, который мы называем *дочерним списком* узла x. Каждый дочерний узел y в дочернем списке имеет указатели $y.left$ и $y.right$, которые указывают на левый и правый "братские" узлы узла y соответственно. Если узел y имеет только один дочерний узел, то $y.left = y.right = y$. Дочерние узлы в дочернем списке могут находиться в любом порядке.

Циклический дважды связанный список (см. раздел 10.2) обладает двумя преимуществами для использования в фибоначчиевых пирамидах. Во-первых, удаление элемента из такого списка выполняется за время $O(1)$. Во-вторых, если имеется два таких списка, их легко объединить в один за то же время $O(1)$. В описании операций над фибоначчиевыми пирамидами мы будем неформально ссылаться на эти операции, оставляя читателю самостоятельно добавить все детали их реализации.

Каждый узел имеет два других атрибута. Мы храним количество дочерних узлов в дочернем списке узла x в $x.degree$. Булев атрибут $x.mark$ указывает, были ли потери узлом x дочерних узлов начиная с момента, когда x стал дочерним узлом какого-то другого узла. Вновь создаваемые узлы не помечены, а имеющаяся

18 Зак. 3726

пометка снимается, если узел становится дочерним узлом какого-то другого узла. До тех пор, пока мы не встретимся с операцией DECREASE-KEY в разделе 19.3, мы просто устанавливаем все атрибуты *mark* равными FALSE.

Обращение к данной фибоначчиевой пирамиде H выполняется посредством указателя $H.min$ на корень дерева, содержащего минимальный ключ. Этот узел называется ***минимальным узлом*** (minimum node) фибоначчиевой пирамиды. Если ключ с минимальным значением имеют более одного корня, то минимальным узлом может служить любой такой корень. Если фибоначчиева пирамида H пуста, то $H.min$ равен NIL.

Корни всех деревьев в фибоначчиевой пирамиде связаны с помощью указателей *left* и *right* в циклический дважды связанный ***список корней*** (root list) фибоначчиевой пирамиды. Указатель $H.min$, таким образом, указывает на узел списка корней, ключ которого минимален. Порядок деревьев в списке корней может быть любым.

Фибоначчиева пирамида, кроме того, имеет еще один атрибут — текущее количество узлов в фибоначчиевой пирамиде H содержится в $H.n$.

Функция потенциала

Как упоминалось, для анализа производительности операций над фибоначчиевыми пирамидами мы будем использовать метод потенциалов из раздела 17.3. Для данной фибоначчиевой пирамиды H обозначим через $t(H)$ количество деревьев в списке корней H, а через $m(H)$ — количество помеченных узлов в H. Тогда потенциал $\Phi(H)$ фибоначчиевой пирамиды H определяется как

$$\Phi(H) = t(H) + 2\,m(H)\ . \qquad (19.1)$$

(Подробнее об этой функции потенциала мы поговорим в разделе 19.3.) Например, потенциал фибоначчиевой пирамиды, показанной на рис. 19.2, равен $5 + 2 \cdot 3 = 11$. Потенциал множества фибоначчиевых пирамид представляет собой сумму потенциалов составляющих его пирамид. Будем считать, что единицы потенциала достаточно для оплаты константного количества работы, где константа достаточно велика для покрытия стоимости любой операции со временем работы $O(1)$.

Кроме того, предполагается, что приложение начинает свою работу, не имея ни одной фибоначчиевой пирамиды, так что начальный потенциал равен 0 и, в соответствии с формулой (19.1), во все последующие моменты времени потенциал неотрицателен. Из (17.3) верхняя граница общей амортизированной стоимости является, таким образом, верхней границей общей фактической стоимости последовательности операций.

Максимальная степень

Амортизационный анализ, который будет выполнен в оставшихся разделах главы, предполагает, что известна верхняя граница $D(n)$ максимальной степени любого узла в фибоначчиевой пирамиде из n узлов. Мы не доказываем этого, но при поддержке только операций над объединяемыми пирамидами $D(n) \leq \lfloor \lg n \rfloor$.

(В упр. 19.2,(г) читателям предлагается самостоятельно доказать это свойство.) В разделах 19.3 и 19.4 мы покажем, что при дополнительной поддержке операций DECREASE-KEY и DELETE $D(n) = O(\lg n)$.

19.2. Операции над объединяемыми пирамидами

Операции над объединяемыми пирамидами в случае фибоначчиевых пирамид откладывают выполнение работы настолько, насколько это возможно. По сути, это компромисс между реализациями различных операций. Например, мы вставляем узел, добавляя его в список корней, что требует только константного времени. Если начать с пустой фибоначчиевой пирамиды и вставить в нее k узлов, то фибоначчиева пирамида будет представлять собой просто список корней из k узлов. Компромисс заключается в том, что если затем мы выполняем над фибоначчиевой пирамидой H операцию EXTRACT-MIN, то после удаления узла, на который указывает $H.min$, мы должны просмотреть каждый из остающихся в списке корней $k - 1$ узлов в поисках нового минимального узла. Раз уж мы все равно должны пройти по всему списку корней в процессе выполнения операции EXTRACT-MIN, мы заодно собираем узлы в неубывающие деревья, чтобы уменьшить размер списка корней. Мы увидим, что независимо от того, как выглядел список корней непосредственно перед выполнением операции EXTRACT-MIN, после нее каждый узел в списке корней имеет степень, уникальную в пределах списка, а это приводит к тому, что размер списка корней не превышает $D(n) + 1$.

Создание фибоначчиевой пирамиды

Для создания пустой фибоначчиевой пирамиды процедура MAKE-FIB-HEAP выделяет память и возвращает объект фибоначчиевой пирамиды H, причем $H.n = 0$ и $H.min =$ NIL; деревьев в H нет. Поскольку $t(H) = 0$ и $m(H) = 0$, потенциал пустой фибоначчиевой пирамиды $\Phi(H) = 0$. Таким образом, амортизированная стоимость процедуры MAKE-FIB-HEAP равна ее фактической стоимости $O(1)$.

Вставка узла

Приведенная далее процедура вставляет узел x в фибоначчиеву пирамиду H в предположении, что узлу уже выделена память и атрибут узла $x.key$ уже заполнен.



Рис. 19.3. Вставка узла в фибоначчиеву пирамиду. **(а)** Фибоначчиева пирамида H. **(б)** Фибоначчиева пирамида H после вставки узла с ключом 21. Узел становится собственным неубывающим деревом и добавляется в список корней, оказываясь левым братом корня.

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6      Создать список корней H, содержащий только x
 7      H.min = x
 8  else Вставить x в список корней H
 9      if x.key < H.min.key
10          H.min = x
11  H.n = H.n + 1
```

В строках 1–4 инициализируются некоторые структурные атрибуты узла x. В строке 5 выполняется проверка, является ли фибоначчиева пирамида H пустой. Если является, то в строках 6 и 7 узел x делается единственным узлом списка корней H, а атрибут $H.min$ получает значение, указывающее на x. В противном случае в строках 8–10 выполняются вставка x в список корней H и при необходимости обновление атрибута $H.min$. Наконец в строке 11 увеличивается значение $H.n$, отражая добавление нового узла. На рис. 19.3 показана вставка узла с ключом 21 в фибоначчиеву пирамиду, приведенную на рис. 19.2.

Для определения амортизированной стоимости FIB-HEAP-INSERT рассмотрим исходную фибоначчиеву пирамиду H и фибоначчиеву пирамиду H', которая получается в результате вставки узла. Тогда $t(H') = t(H) + 1$ и $m(H') = m(H)$, и увеличение потенциала составляет

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 .$$

Поскольку фактическая стоимость равна $O(1)$, амортизированная стоимость равна $O(1) + 1 = O(1)$.

Поиск минимального узла

На минимальный узел фибоначчиевой пирамиды H указывает указатель $H.min$, так что поиск минимального узла занимает время $O(1)$. Поскольку по-

тенциал H при этом не изменяется, амортизированная стоимость этой операции равна ее фактической стоимости $O(1)$.

Объединение двух фибоначчиевых пирамид

Приведенная далее процедура объединяет фибоначчиевы пирамиды H_1 и H_2, уничтожая их в процессе объединения. Процедура попросту соединяет списки корней H_1 и H_2 и находит затем новый минимальный узел. По окончании работы объекты, представляющие H_1 и H_2, далее использоваться не могут.

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  Конкатенация списков корней H2 и H
4  if (H1.min == NIL) или (H2.min ≠ NIL и H2.min.key < H1.min.key)
5      H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

В строках 1–3 выполняется конкатенация списков корней фибоначчиевых пирамид H_1 и H_2 в новый список корней фибоначчиевой пирамиды H. Строки 2, 4 и 5 устанавливают минимальный узел фибоначчиевой пирамиды H, а в строке 6 определяется общее количество узлов $H.n$. Возврат полученной фибоначчиевой пирамиды H выполняется в строке 7. Как и в процедуре FIB-HEAP-INSERT, все корни остаются корнями.

Изменение потенциала составляет

$$
\begin{aligned}
\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
&= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
&= 0\,,
\end{aligned}
$$

поскольку $t(H) = t(H_1) + t(H_2)$ и $m(H) = m(H_1) + m(H_2)$. Таким образом, амортизированная стоимость FIB-HEAP-UNION равна ее фактической стоимости $O(1)$.

Извлечение минимального узла

Процесс извлечения минимального узла наиболее сложный из всех операций, рассматриваемых в данном разделе. Это также то место, где выполняются отложенные действия по объединению деревьев. Псевдокод процедуры извлечения минимального узла приведен ниже. Для удобства предполагается, что, когда узел удаляется из связанного списка, указатели, остающиеся в списке, обновляются, но указатели в извлекаемом узле остаются неизменными. В этой процедуре используется вспомогательная процедура CONSOLIDATE, которая будет рассмотрена чуть позже.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3      for каждый дочерний узел x узла z
 4          Добавить x в список корней H
 5          x.p = NIL
 6      Удалить z из списка корней H
 7      if z == z.right
 8          H.min = NIL
 9      else H.min = z.right
10          CONSOLIDATE(H)
11      H.n = H.n − 1
12  return z
```

Как показано на рис. 19.4, процедура FIB-HEAP-EXTRACT-MIN сначала создает список корней из дочерних узлов минимального узла, а затем удаляет последний из списка корней. Затем выполняется уплотнение списка корней путем связывания корней одинаковой степени, пока в списке останется не больше одного корня каждой степени.

Работа начинается в строке 1, где в переменной z сохраняется указатель на минимальный узел фибоначчиевой пирамиды, — именно этот указатель процедура вернет в конце работы. Если $z =$ NIL, это означает, что фибоначчиева пирамида H пуста, и работа на этом заканчивается. В противном случае мы удаляем узел z из H, делая все дочерние узлы узла z корнями в H в строках 3–5 (помещая их в список корней) и вынося из него z в строке 6. Если после выполнения строки 6 z является собственным правым братом, значит, z был единственным узлом в списке корней, и у него нет дочерних узлов. В этом случае нам остается только сделать фибоначчиеву пирамиду пустой в строке 8, перед тем как вернуть z. В противном случае мы устанавливаем указатель $H.min$ на корень, отличный от z (в нашем случае это правый брат z), причем это не обязательно минимальный узел по завершении процедуры FIB-HEAP-EXTRACT-MIN. На рис. 19.4, (б) показано состояние исходной фибоначчиевой пирамиды, приведенной на рис. 19.4, (а), после выполнения строки 9.

Следующий этап, на котором будет уменьшено количество деревьев в фибоначчиевой пирамиде, — ***уплотнение*** (consolidating) списка корней H, которое выполняется вспомогательной процедурой CONSOLIDATE(H). Уплотнение списка корней состоит в многократном выполнении следующих шагов до тех пор, пока все корни в списке корней не будут иметь различные значения атрибута *degree*.

1. Найти в списке корней два корня x и y с одинаковой степенью. Пусть, без потери общности, $x.key \leq y.key$.

2. ***Связать*** (link) y с x: удалить y из списка корней и сделать его дочерним узлом x. Эта операция выполняется процедурой FIB-HEAP-LINK. Данная процедура увеличивает атрибут $x.degree$ и снимает пометку с y.

Процедура CONSOLIDATE использует вспомогательный массив $A[0\,..\,D(H.n)]$ для отслеживания корней в соответствии с их степенями. Если $A[i] = y$, то y в настоящий момент является корнем со степенью $y.degree = i$. Конечно, для выделения массива необходимо знать верхнюю границу $D(H.n)$ максимальной степени. Как ее вычислить, вы узнаете из раздела 19.4.

```
CONSOLIDATE(H)
 1  Пусть A[0..D(H.n)] — новый массив
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for каждый узел w в списке корней H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]       // Другой узел с той же степенью, что и у x
 9          if x.key > y.key
10              Обменять x и y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              Создать список корней H, содержащий только A[i]
20              H.min = A[i]
21          else Вставить A[i] в список корней H
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  Удалить y из списка корней H
2  Сделать y дочерним узлом x, увеличивая x.degree
3  y.mark = FALSE
```

Рассмотрим процедуру CONSOLIDATE более подробно. В строках 1–3 выполняется инициализация массива A путем присвоения каждому элементу массива значения NIL. Цикл **for** в строках 4–14 обрабатывает каждый корень w из списка корней. При связывании корней w может оказаться связанным с некоторым другим узлом и перестать быть корнем. Тем не менее w всегда находится в дереве, корнем которого является некоторый узел x, который может как совпадать, так и не совпадать с w. Поскольку мы хотим иметь не более одного корня для каждой степени, мы просматриваем массив A, чтобы узнать, не содержит ли он корень y с той же степенью, что и у x. Если это так, то мы связываем корни x и y, но

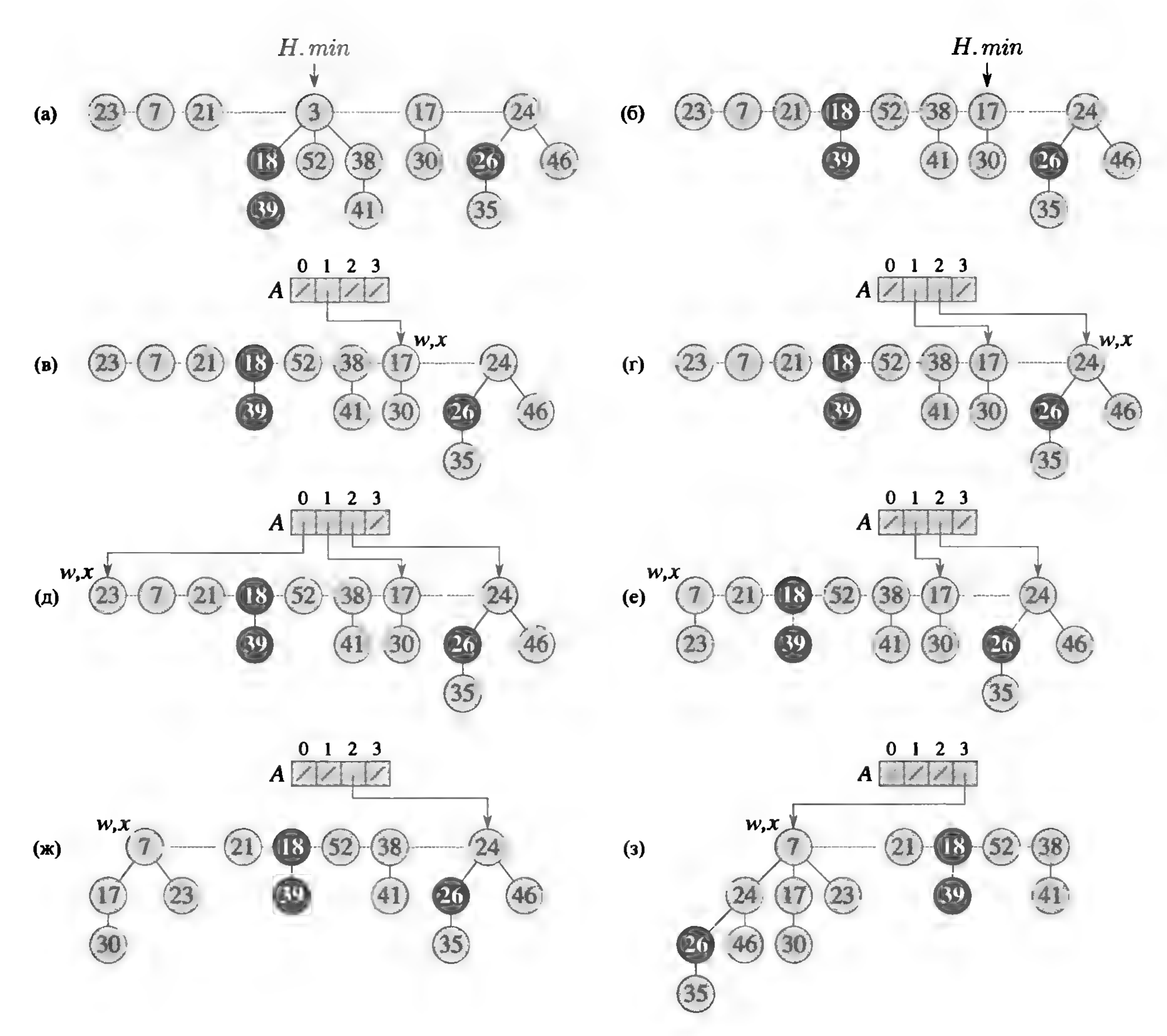


Рис. 19.4. Работа процедуры FIB-HEAP-EXTRACT-MIN. **(а)** Фибоначчиева пирамида H. **(б)** Ситуация после удаления минимального узла z из списка корней и добавления в список его дочерних узлов. **(в)–(д)** Массив A и деревья после каждой из трех итераций цикла **for** в строках 4–14 процедуры CONSOLIDATE. Процедура обрабатывает список корней начиная с узла, на который указывает $H.min$, и следуя указателям $right$. В каждой части показаны значения w и x в конце итерации. **(е)–(з)** Следующая итерация цикла **for** с указанием значений w и x в конце каждой итерации цикла **while** в строках 7–13. В части (е) показана ситуация после первого прохода цикла **while**. Узел с ключом 23 связан с узлом с ключом 7, на который в этот момент указывает x. В части (ж) узел с ключом 17 был связан с узлом с ключом 7, на который все еще указывает x. В части (з) узел с ключом 24 был связан с узлом с ключом 7. Поскольку ранее $A[3]$ не указывал ни на один узел, в конце итерации цикла **for** значение $A[3]$ устанавливается указывающим на корень полученного в результате дерева.

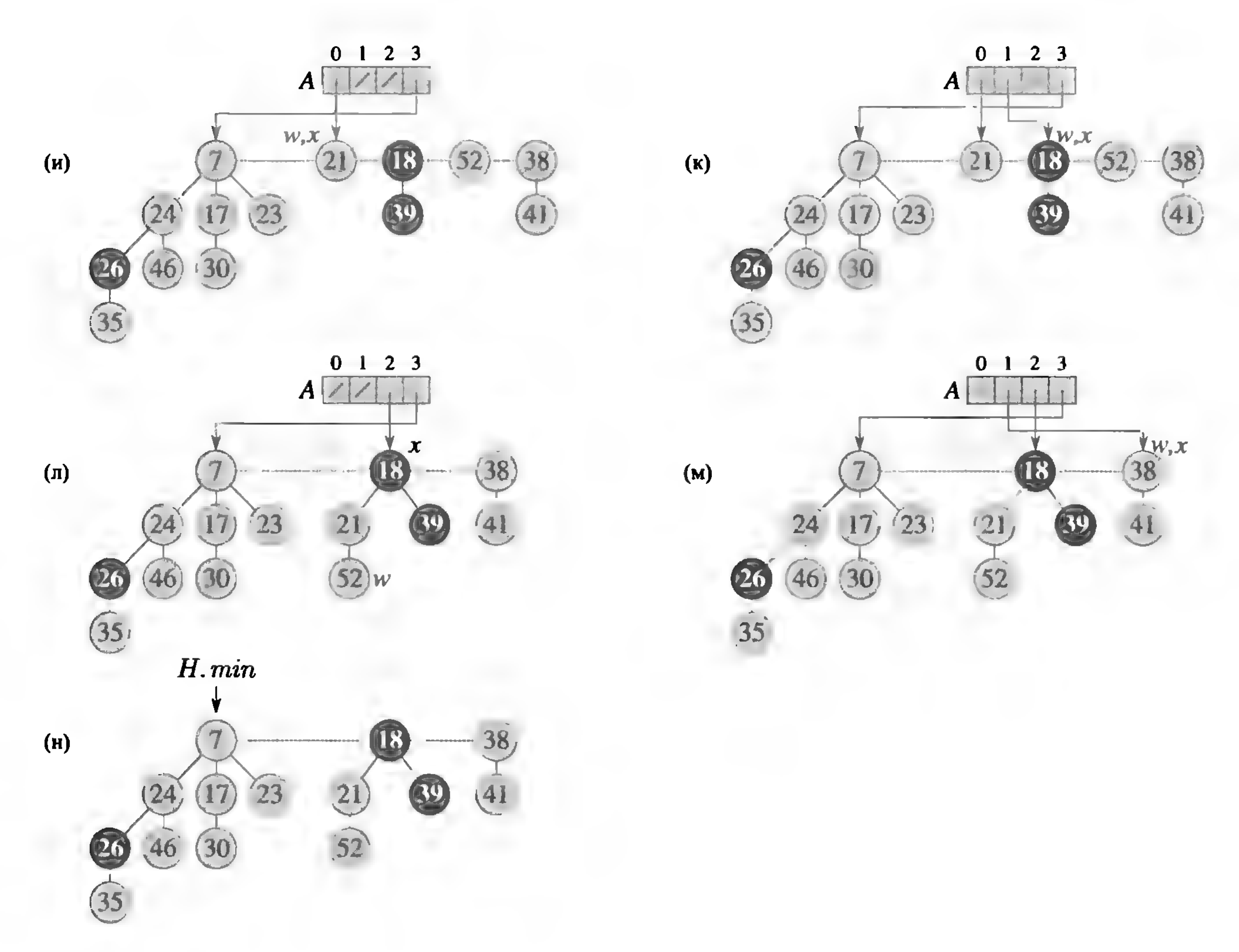


Рис. 19.4 (продолжение). **(и)–(м)** Ситуация после выполнения каждой из следующих четырех итераций цикла **for**. **(н)** Фибоначчиева пирамида H после восстановления списка корней из массива A и определения нового указателя $H.min$.

гарантируем, что после этого x останется корнем. Иначе говоря, мы связываем y с x после первого обмена указателей на эти корни, если ключ y меньше ключа x. После связывания y с x степень x увеличивается на 1, так что мы продолжаем процесс, связывая x и другой корень, степень которого равна новой степени x, и так до тех пор, пока больше не будет корней с той же степенью, что и у x. Затем соответствующий элемент массива A делается указывающим на x, так что при последующей обработке корней у нас уже будет записано, что x является единственным корнем с данной степенью, который уже обработан. По завершении работы цикла **for** в списке корней останется не более чем по одному корню каждой степени, и элементы массива A будут указывать на каждый из оставшихся в списке корней.

Цикл **while** в строках 7–13 связывает корень x дерева, в котором содержится узел w, с другим деревом, корень которого имеет ту же степень, что и x. Это действие повторяется до тех пор, пока ни один другой корень не будет иметь ту же степень, что и x. Инвариант цикла **while** следующий:

В начале каждой итерации цикла **while** $d = x.degree$.

Воспользуемся этим инвариантом цикла для доказательства корректности алгоритма.

Инициализация. Строка 6 гарантирует, что инвариант цикла выполняется при первом входе в цикл.

Сохранение. В каждой итерации цикла **while** $A[d]$ указывает на некоторый корень y. Поскольку $d = x.degree = y.degree$, мы хотим связать x и y. В результате операции связывания тот из узлов x и y, который имеет меньший ключ, становится родителем другого, так что в строках 9 и 10 при необходимости выполняется обмен указателей на x и y. Затем мы связываем y с x вызовом FIB-HEAP-LINK(H, y, x) в строке 11. Этот вызов увеличивает $x.degree$, но оставляет $y.degree$ равным d. Узел y больше не является корнем, так что в строке 12 указатель на него удаляется из массива A. Поскольку вызов FIB-HEAP-LINK увеличивает значение $x.degree$, строка 13 восстанавливает инвариант $d = x.degree$.

Завершение. Цикл **while** повторяется до тех пор, пока не будет выполнено условие $A[d] =$ NIL, т.е. пока не будет другого корня с той же степенью, что и у x.

После завершения цикла **while** мы присваиваем элементу $A[d]$ значение x в строке 14 и выполняем очередную итерацию цикла **for**.

На рис. 19.4,(в)–(д) показаны массив A и деревья, которые получаются в результате первых трех итераций цикла **for** в строках 4–14. В следующей итерации цикла **for** выполняются три связывания; их результаты можно увидеть на рис. 19.4,(е)–(з). На рис. 19.4,(и)–(м) представлены результаты следующих четырех итераций цикла **for**.

Теперь все, что остается, — выполнить завершающие действия. После того как закончена работа цикла в строках 4–14, строка 15 создает пустой список, который заполняется в строках 16–23 на основе данных из массива A. Полученная в результате фибоначчиева пирамида показана на рис. 19.4,(н). После уплотнения списка корней процедура FIB-HEAP-EXTRACT-MIN завершается уменьшением $H.n$ в строке 11 и возвратом указателя на удаленный узел z в строке 12.

Теперь мы готовы показать, что амортизированная стоимость извлечения минимального узла из фибоначчиевой пирамиды с n узлами равна $O(D(n))$. Обозначим через H фибоначчиеву пирамиду непосредственно перед выполнением операции FIB-HEAP-EXTRACT-MIN.

Начнем с подсчета фактической стоимости извлечения минимального узла. Вклад $O(D(n))$ вносит обработка не более $D(n)$ дочерних узлов минимального узла в процедуре FIB-HEAP-EXTRACT-MIN, а также операции в строках 2 и 3 и 16–23 процедуры CONSOLIDATE. Остается проанализировать вклад цикла **for** в строках 4–14 той же процедуры, для чего применим амортизационный анализ. Размер списка корней при вызове процедуры CONSOLIDATE не превышает $D(n) + t(H) - 1$, поскольку он состоит из исходного списка корней с $t(H)$ узлами, минус извлеченный узел и плюс дочерние узлы извлеченного узла, количество которых не превышает $D(n)$. В заданной итерации цикла **for** в строках 4–14 количество итераций цикла **while** в строках 7–13 зависит от списка корней. Но мы знаем, что при каждом проходе цикла **while** один из корней связывается с другим, так что общее количество итераций цикла **while** во всех итерациях цикла **for** не превышает количества элементов списка корней. Следовательно, об-

щее количество работы, выполняемой в цикле **for**, не более чем пропорционально $D(n)+t(H)$. Таким образом, фактическая работа по извлечению минимального узла равна $O(D(n) + t(H))$.

Потенциал перед извлечением минимального узла равен $t(H) + 2m(H)$, а после – не превышает $(D(n) + 1) + 2m(H)$, поскольку остается не более $D(n) + 1$ корней, и нет узлов, которые были бы помечены во время выполнения этой операции. Таким образом, амортизированная стоимость не превосходит величину

$$O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) \\ = O(D(n)) + O(t(H)) - t(H) \\ = O(D(n)) \ ,$$

поскольку можно масштабировать единицы потенциала таким образом, чтобы можно было пренебречь константой, скрытой в $O(t(H))$. Интуитивно понятно, что стоимость выполнения каждого связывания является платой за снижение потенциала из-за уменьшения количества корней на 1 при связывании. В разделе 19.4 мы увидим, что $D(n) = O(\lg n)$, так что амортизированная стоимость извлечения минимального узла равна $O(\lg n)$.

Упражнения

19.2.1

Изобразите фибоначчиеву пирамиду, которая получается в результате применения процедуры FIB-HEAP-EXTRACT-MIN к фибоначчиевой пирамиде, показанной на рис. 19.4, (н).

19.3. Уменьшение ключа и удаление узла

В этом разделе будет показано, как уменьшить ключ узла фибоначчиевой пирамиды за амортизированное время $O(1)$ и как удалить произвольный узел из фибоначчиевой пирамиды с n узлами за амортизированное время $O(D(n))$. В разделе 19.4 будет показано, что максимальная степень $D(n)$ равна $O(\lg n)$, откуда вытекает, что амортизированное время работы процедур FIB-HEAP-EXTRACT-MIN и FIB-HEAP-DELETE составляет $O(\lg n)$.

Уменьшение ключа

В приведенном далее псевдокоде операции FIB-HEAP-DECREASE-KEY, как и ранее, предполагается, что при удалении узла из связанного списка никакие его структурные атрибуты не изменяются.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "Новый ключ больше текущего"
3  x.key = k
4  y = x.p
5  if y ≠ NIL и x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x
```

```
CUT(H, x, y)
1  Удалить x из дочернего списка y, уменьшая y.degree
2  Добавить x в список корней H
3  x.p = NIL
4  x.mark = FALSE
```

```
CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3      if y.mark == FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```

Процедура FIB-HEAP-DECREASE-KEY работает следующим образом. В строках 1–3 проверяется, не окажется ли новый ключ больше старого, и если нет, то x получает новое значение ключа. Если x — корень или если $x.key \geq y.key$, где y — родитель x, то никакие структурные изменения не нужны, поскольку свойство неубывающих пирамид не нарушено. Это условие проверяется в строках 4 и 5.

Если же свойство неубывающих пирамид оказывается нарушенным, может потребоваться внести множество изменений. Мы начинаем с ***вырезания*** (cutting) x в строке 6. Процедура CUT "вырезает" связь между x и его родительским узлом y, делая x корнем.

Атрибуты *mark* используются для получения желаемого времени работы. В них хранится маленькая часть истории каждого узла. Предположим, что с узлом x произошли следующие события.

1. В некоторый момент x был корнем.
2. x был связан с другим узлом (стал его дочерним узлом).
3. Два дочерних узла x были вырезаны.

Как только x теряет второй дочерний узел, мы вырезаем x у его родителя, делая x новым корнем. Атрибут $x.mark$ равен TRUE, если произошли события 1

и 2 и у x вырезан только один дочерний узел. Процедура CUT, таким образом, в строке 4 очищает атрибут $x.mark$, поскольку произошло событие 1. (Теперь становится понятно, почему в строке 3 процедуры FIB-HEAP-LINK выполняется сброс $y.mark$: узел y оказывается связанным с другим узлом, т.е. выполняется событие 2. Затем, когда будет вырезаться дочерний узел у узла y, $y.mark$ получит значение TRUE.)

Выполнена еще не вся работа, поскольку x может быть вторым дочерним узлом, вырезанным у его родительского узла y с того момента, когда y был привязан к другому узлу. Поэтому в строке 7 процедуры FIB-HEAP-DECREASE-KEY предпринимается попытка выполнить операцию ***каскадного вырезания*** (cascading-cut) над y. Если y — корень, то проверка в строке 2 процедуры CASCADING-CUT заставляет последнюю прекратить работу. Если y не помечен, процедура помечает его в строке 4, поскольку его первый дочерний узел был только что вырезан, после чего также прекращает работу. Однако если узел уже был помечен, значит, только что он потерял второй дочерний узел. Тогда y вырезается в строке 5, и процедура CASCADING-CUT в строке 6 рекурсивно вызывает себя для узла z, родительского по отношению к узлу y. Процедура CASCADING-CUT рекурсивно поднимается вверх по дереву до тех пор, пока не достигает корня или непомеченного узла.

После того как выполнены все каскадные вырезания, в строках 8 и 9 процедуры FIB-HEAP-DECREASE-KEY при необходимости обновляется величина $H.min$. Единственным узлом, ключ которого изменяется в процессе работы процедуры, является узел x, так что минимальным узлом может быть только исходный минимальный узел или узел x.

На рис. 19.5 показано выполнение двух вызовов FIB-HEAP-DECREASE-KEY над фибоначчиевой пирамидой, приведенной на рис. 19.5, (а). Первый вызов, показанный на рис. 19.5, (б), выполняется без каскадного вырезания. При втором вызове (рис. 19.5, (в)–(д)) выполняются два каскадных вырезания.

Покажем теперь, что амортизированная стоимость процедуры FIB-HEAP-DECREASE-KEY составляет лишь $O(1)$. Начнем с определения фактической стоимости. Процедура FIB-HEAP-DECREASE-KEY выполняется за время $O(1)$, плюс время, необходимое для каскадного вырезания. Предположим, что процедура CASCADING-CUT в данном вызове FIB-HEAP-DECREASE-KEY рекурсивно вызывается c раз (вызов выполняется в строке 7 процедуры FIB-HEAP-DECREASE-KEY, за которым следует $c - 1$ рекурсивный вызов CASCADING-CUT). Каждый вызов CASCADING-CUT без учета рекурсии требует времени $O(1)$. Следовательно, фактическая стоимость процедуры FIB-HEAP-DECREASE-KEY с учетом всех рекурсивных вызовов составляет $O(c)$.

Вычислим теперь изменение потенциала. Обозначим через H фибоначчиеву пирамиду непосредственно перед вызовом процедуры FIB-HEAP-DECREASE-KEY. Вызов CUT в строке 6 процедуры FIB-HEAP-DECREASE-KEY создает новое дерево с корнем в узле x и сбрасывает бит метки (который уже может иметь значение FALSE). Каждый вызов CASCADING-CUT, за исключением последнего, вырезает помеченный узел и сбрасывает бит метки. После этого в фибоначчиевой пирамиде имеется $t(H)+c$ деревьев (исходные $t(H)$ деревьев, $c-1$ деревьев, по-

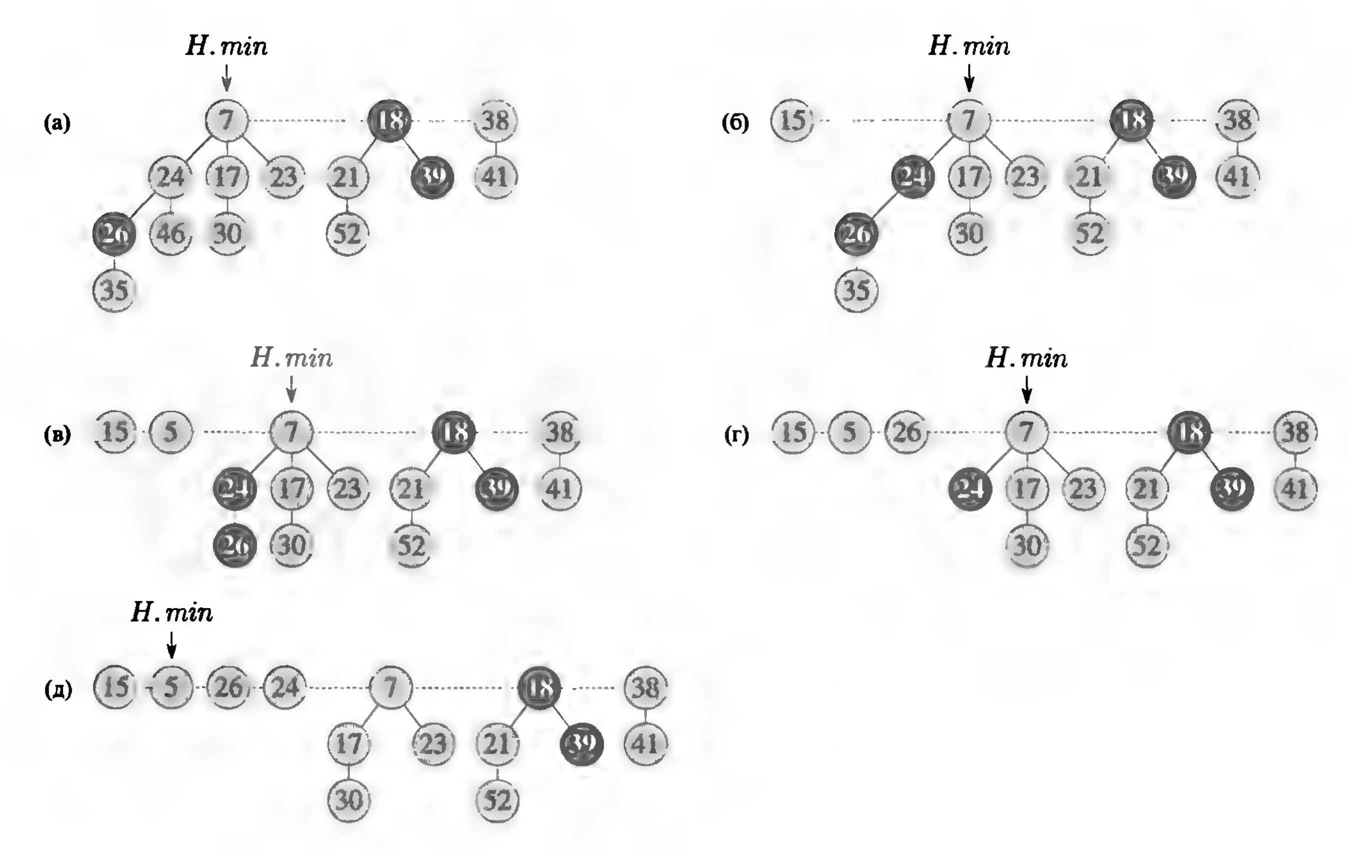


Рис. 19.5. Два вызова FIB-HEAP-DECREASE-KEY. **(а)** Исходная фибоначчиева пирамида. **(б)** Ключ узла уменьшается с 46 до 15. Узел становится корнем, а его родительский узел (с ключом 24), который ранее был непомеченным, помечается. **(в)–(д)** Ключ узла уменьшается с 35 до 5. В части (в) узел, теперь имеющий ключ 5, становится корнем. Его родительский узел, с ключом 26, помечен, так что осуществляется каскадное вырезание. Узел с ключом 26 вырезается у родителя и делается непомеченным корнем в части (г). Выполняется и другое каскадное вырезание, поскольку узел с ключом 24 также помечен. Этот узел вырезается у родителя и делается непомеченным корнем в части (д). В этот момент каскадные вырезания прекращаются, поскольку узел с ключом 7 является корнем. (Даже если бы этот узел не был корнем, каскадные вырезания остановились бы, так как он непомеченный.) В части (д) показан результат выполнения операции FIB-HEAP-DECREASE-KEY, с $H.min$, указывающим на новый минимальный узел.

лученных при каскадном вырезании, и дерево, корнем которого является x) и не более $m(H)-c+2$ помеченных узла (у $c-1$ узлов пометка была снята при каскадном вырезании, а последний вызов процедуры CASCADING-CUT может привести к пометке узла). Таким образом, изменение потенциала не превышает величины

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2\,m(H))=4-c\;.$$

Таким образом, амортизированная стоимость FIB-HEAP-DECREASE-KEY не превышает

$$O(c)+4-c=O(1)\;,$$

поскольку мы можем соответствующим образом масштабировать единицы потенциала для доминирования над константой, скрытой в $O(c)$.

Теперь вы видите, почему функция потенциала определена таким образом, что включает член, равный удвоенному количеству помеченных узлов. Когда помеченный узел y вырезается в процессе каскадного вырезания, его пометка сни-

мается, что приводит к снижению потенциала на 2. Одна единица потенциала служит платой за вырезание и снятие пометки, а вторая компенсирует увеличение потенциала из-за того, что y становится корнем.

Удаление узла

Приведенный далее псевдокод выполняет удаление узла из фибоначчиевой пирамиды с n узлами за амортизированное время $O(D(n))$. Предполагается, что в настоящий момент в фибоначчиевой пирамиде нет ключа со значением $-\infty$.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

Процедура FIB-HEAP-DELETE делает x минимальным узлом фибоначчиевой пирамиды, присваивая ему уникальное минимальное значение ключа $-\infty$. Затем процедура FIB-HEAP-EXTRACT-MIN удаляет узел x из фибоначчиевой пирамиды. Амортизированное время работы FIB-HEAP-DELETE представляет собой сумму амортизированного времени работы $O(1)$ процедуры FIB-HEAP-DECREASE-KEY и амортизированного времени работы $O(D(n))$ процедуры FIB-HEAP-EXTRACT-MIN. Поскольку в разделе 19.4 мы увидим, что $D(n) = O(\lg n)$, амортизированное время работы процедуры FIB-HEAP-DELETE составляет $O(\lg n)$.

Упражнения

19.3.1

Предположим, что корень x в фибоначчиевой пирамиде помечен. Поясните, как узел x мог стать помеченным корнем. Покажите, что тот факт, что x помечен, не имеет никакого значения для анализа, даже если это не корень, который сначала был привязан к другому узлу, а затем потерял один дочерний узел.

19.3.2

Докажите оценку $O(1)$ амортизированного времени работы процедуры FIB-HEAP-DECREASE-KEY как средней стоимости операции с использованием группового анализа.

19.4. Оценка максимальной степени

Для доказательства того факта, что амортизированное время работы процедур FIB-HEAP-EXTRACT-MIN и FIB-HEAP-DELETE равно $O(\lg n)$, необходимо показать, что верхняя граница $D(n)$ степени произвольного узла в фибоначчиевой пирамиде с n узлами равна $O(\lg n)$. В частности, следует показать, что $D(n) \le \lfloor \log_\phi n \rfloor$, где ϕ — золотое сечение, определяемое формулой (3.24) как

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots .$$

Ключевым моментом анализа является следующее. Для каждого узла x в фибоначчиевой пирамиде определим $\text{size}(x)$ как количество узлов в поддереве, корнем которого является x, включая сам узел x (заметим, что узел x не обязательно должен находиться в списке корней; это может быть любой узел фибоначчиевой пирамиды). Покажем, что величина $\text{size}(x)$ экспоненциально зависит от $x.degree$ (напомним, что атрибут $x.degree$ всегда содержит точное значение степени x).

Лемма 19.1

Пусть x — произвольный узел фибоначчиевой пирамиды, и предположим, что $x.degree = k$. Обозначим через $y_1, y_2, \ldots, y_k$ дочерние узлы x в том порядке, в котором они связаны с x, начиная с более ранних и заканчивая более поздними. Тогда $y_1.degree \geq 0$ и $y_i.degree \geq i - 2$ для $i = 2, 3, \ldots, k$.

Доказательство. Очевидно, $y_1.degree \geq 0$.

Для $i \geq 2$ заметим, что, когда y_i связывается с x, все узлы $y_1, y_2, \ldots, y_{i-1}$ являются дочерними узлами x, так что в этот момент должно выполняться $x.degree \geq i - 1$. Узел y_i связывается процедурой CONSOLIDATE с x только в случае, когда $x.degree = y_i.degree$, так что в этот момент мы должны также иметь $y_i.degree \geq i - 1$. С этого момента узел y_i мог потерять не более одного дочернего узла, поскольку при потере двух дочерних узлов он должен быть вырезан у узла x процедурой CASCADING-CUT. Отсюда следует, что $y_i.degree \geq i - 2$. ∎

Сейчас мы подошли к той части анализа, которая поясняет название "фибоначчиевы пирамиды". Вспомним, что в разделе 3.2 k-е число Фибоначчи ($k = 0, 1, 2, \ldots$) определяется с помощью рекуррентного соотношения

$$F_k = \begin{cases} 0\,, & \text{если } k = 0\,, \\ 1\,, & \text{если } k = 1\,, \\ F_{k-1} + F_{k-2}\,, & \text{если } k \geq 2\,. \end{cases}$$

Приведенная далее лемма дает еще один способ выражения F_k.

Лемма 19.2

Для всех целых чисел $k \geq 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i\,.$$

Доказательство. Доказательство выполняется по индукции по k. Когда $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= F_2\,. \end{aligned}$$

По индукции предполагаем, что $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, и имеем

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i \ . \end{aligned}$$

■

Лемма 19.3

$(k+2)$-е число Фибоначчи для всех целых $k \geq 0$ удовлетворяет условию $F_{k+2} \geq \phi^k$.

Доказательство. Докажем лемму с помощью математической индукции по k. Базой индукции служат значения $k = 0$ и $k = 1$. При $k = 0$ мы имеем $F_2 = 1 = \phi^0$, а при $k = 1$ – $F_3 = 2 > 1.619 > \phi^1$. Шаг индукции выполняется для $k \geq 2$, и мы считаем, что $F_{i+2} > \phi^i$ для $i = 0, 1, \ldots, k-1$. Вспомним, что ϕ – положительный корень уравнения (3.23), $x^2 = x + 1$. Таким образом, мы имеем

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(согласно гипотезе индукции)} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 && \text{(согласно уравнению (3.23))} \\ &= \phi^k \ . \end{aligned}$$

■

Следующая лемма и следствие из нее завершают анализ.

Лемма 19.4

Пусть x – произвольный узел фибоначчиевой пирамиды и пусть $k = x.degree$. Тогда $\text{size}(x) \geq F_{k+2} \geq \phi^k$, где $\phi = (1 + \sqrt{5})/2$.

Доказательство. Обозначим через s_k минимально возможный размер любого узла степени k в произвольной фибоначчиевой пирамиде. Случаи $s_0 = 1$ и $s_1 = 2$ тривиальны. Число s_k не превышает величины $\text{size}(x)$ и, поскольку добавление дочерних узлов к узлу не может уменьшить его размер, значение s_k монотонно возрастает с возрастанием k. Рассмотрим некоторый узел z в произвольной фибоначчиевой пирамиде, такой, что $z.degree = k$ и $\text{size}(z) = s_k$. Так как $s_k \leq \text{size}(x)$, мы вычисляем нижнюю границу $\text{size}(x)$ путем вычисления нижней границы s_k. Как и в лемме 19.1, обозначим через $y_1, y_2, \ldots, y_k$ дочерние узлы z в порядке их связывания с z. При вычислении нижней границы s_k учтем по единице для

самого́ z и для его первого дочернего узла y_1 (для которого $\text{size}(y_1) \geq 1$). Тогда

$$\begin{aligned}
\text{size}(x) &\geq s_k \\
&\geq 2 + \sum_{i=2}^{k} s_{y_i.\mathit{degree}} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} ,
\end{aligned}$$

где последний переход следует из леммы 19.1 (так что $y_i.\mathit{degree} \geq i - 2$) и монотонности s_k (так что $s_{y_i.\mathit{degree}} \geq s_{i-2}$).

Покажем теперь по индукции по k, что $s_k \geq F_{k+2}$ для всех неотрицательных целых k. База индукции при $k = 0$ и $k = 1$ доказывается тривиально. В качестве шага индукции мы предполагаем, что $k \geq 2$ и что $s_i \geq F_{i+2}$ при $i = 0, 1, \ldots, k-1$. Тогда

$$\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=0}^{k} F_i \\
&= F_{k+2} && \text{(согласно лемме 19.2)} \\
&\geq \phi^k && \text{(согласно лемме 19.3)} .
\end{aligned}$$

Таким образом, мы показали, что $\text{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

Следствие 19.5

Максимальная степень $D(n)$ произвольного узла в фибоначчиевой пирамиде с n узлами равна $O(\lg n)$.

Доказательство. Пусть x — произвольный узел в фибоначчиевой пирамиде с n узлами и пусть $k = x.\mathit{degree}$. Согласно лемме 19.4 $n \geq \text{size}(x) \geq \phi^k$. Логарифмируя по основанию ϕ, получаем $k \leq \log_\phi n$ (в действительности, так как k — целое число, $k \leq \lfloor \log_\phi n \rfloor$). Таким образом, максимальная степень $D(n)$ произвольного узла равна $O(\lg n)$. ■

Упражнения

19.4.1

Профессор утверждает, что высота фибоначчиевой пирамиды из n узлов равна $O(\lg n)$. Покажите, что профессор ошибается и что для любого положительно-

го n имеется последовательность операций, которая создает фибоначчиеву пирамиду, состоящую из одного дерева, которое представляет собой линейную цепочку узлов.

19.4.2

Предположим, что мы обобщили правило каскадного вырезания, и что узел x вырезается у родительского узла, как только теряет k-й дочерний узел, где k — некоторая константа (в правиле из раздела 19.3 использовано значение $k = 2$). Для каких значений k справедливо соотношение $D(n) = O(\lg n)$?

Задачи

19.1. Альтернативная реализация удаления

Профессор Пизано предложил следующий вариант процедуры FIB-HEAP-DELETE, утверждая, что он работает быстрее для случая удаления узла, не являющегося тем, на который указывает $H.min$.

```
PISANO-DELETE(H, x)
1  if x == H.min
2      FIB-HEAP-EXTRACT-MIN(H)
3  else y = x.p
4      if y ≠ NIL
5          CUT(H, x, y)
6          CASCADING-CUT(H, y)
7      Добавить дочерний список x к списку корней H
8      Удалить x из списка корней H
```

а. Утверждение профессора о быстрой работе процедуры частично основано на предположении, что строка 7 может быть выполнена за время $O(1)$. Что неверно в этом предположении?

б. Оцените верхнюю границу фактического времени работы процедуры PISANO-DELETE, когда x не является $H.min$. Ваша оценка должна быть выражена через $x.degree$ и количество c вызовов процедуры CASCADING-CUT.

в. Предположим, что мы вызываем PISANO-DELETE(H, x) и что H' — полученная в результате фибоначчиева пирамида. Считая, что x не является корнем, оцените потенциал H', выразив его через $x.degree$, c, $t(H)$ и $m(H)$.

г. Сделайте вывод о том, что амортизированное время работы процедуры PISANO-DELETE асимптотически не лучше времени работы процедуры FIB-HEAP-DELETE, даже когда $x \neq H.min$.

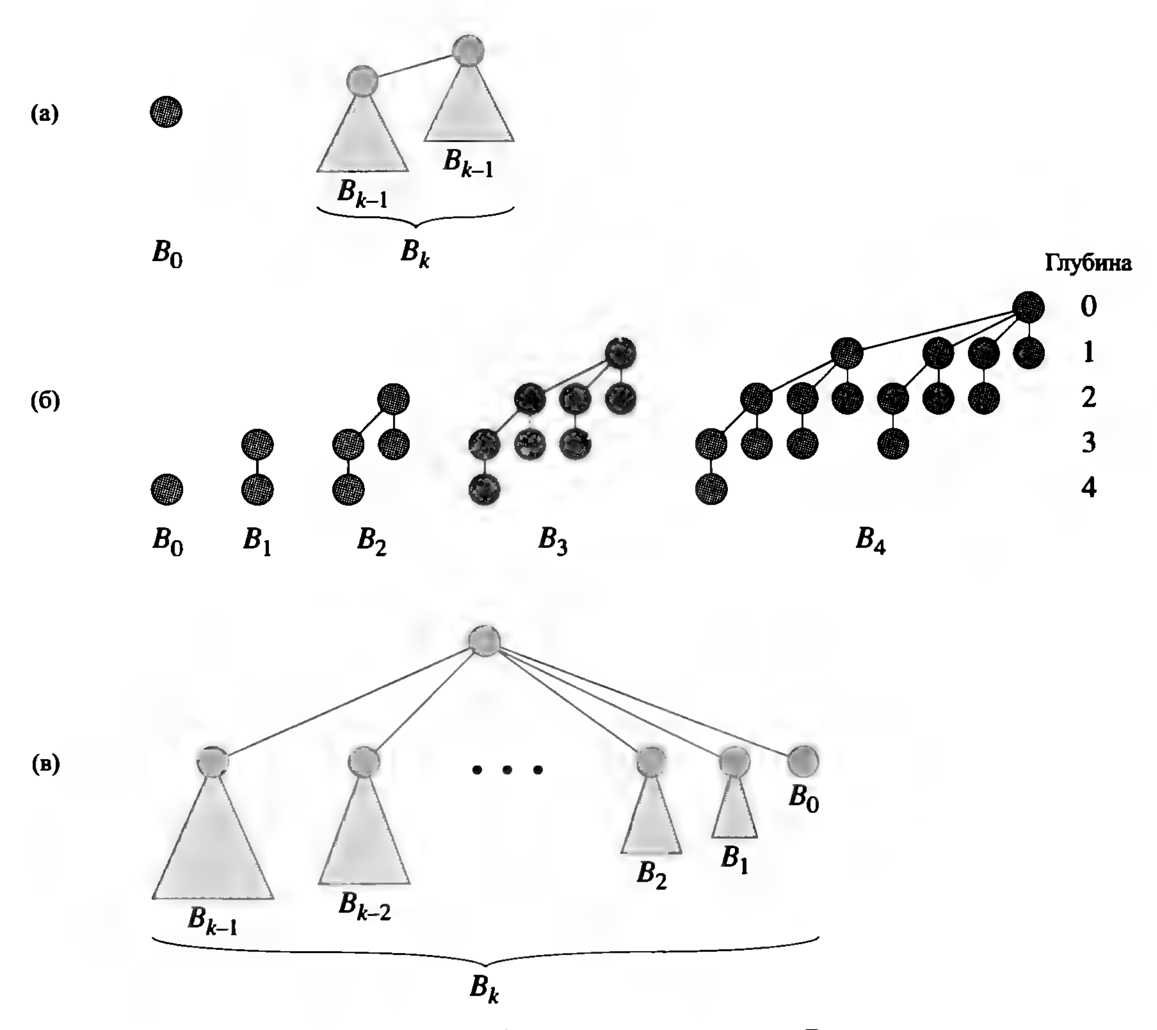


Рис. 19.6. **(а)** Рекурсивное определение биномиального дерева B_k. Треугольники представляют корневые поддеревья. **(б)** Биномиальные деревья от B_0 до B_4. Показана глубина узлов в B_4. **(в)** Другой способ рассмотрения биномиального дерева B_k.

19.2. Биномиальные деревья и биномиальные пирамиды

Биномиальное дерево (binomial tree) B_k представляет собой рекурсивно определенное упорядоченное дерево (см. раздел Б.5.2). Как показано на рис. 19.6, (а), биномиальное дерево B_0 состоит из единственного узла. Биномиальное дерево B_k состоит из двух биномиальных деревьев B_{k-1}, связанных вместе, так что корень одного является крайним слева дочерним узлом корня другого. На рис. 19.6, (б) показаны биномиальные деревья от B_0 до B_4.

а. Покажите, что для биномиального дерева B_k справедливы следующие утверждения:

1. в нем имеется 2^k узлов;
2. его высота равна k;
3. на глубине i ($i = 0, 1, \ldots, k$) в нем находится ровно $\binom{k}{i}$ узлов;
4. корень дерева имеет степень k, которая больше степени любого другого узла; кроме того, как показано на рис. 19.6, (в), если перенумеровать дочерние узлы корня слева направо как $k-1, k-2, \ldots, 0$, то дочерний узел i будет корнем поддерева B_i.

Биномиальная пирамида (binomial heap) H представляет собой множество биномиальных деревьев, удовлетворяющее следующим свойствам.

1. Каждый узел имеет атрибут key (как и фибоначчиева пирамида).
2. Каждое биномиальное дерево в H подчиняется свойству неубывающей пирамиды.
3. Для любого неотрицательного целого числа k в H имеется не более одного биномиального дерева, корень которого имеет степень k.

б. Предположим, что биномиальная пирамида H всего имеет n узлов. Рассмотрите взаимосвязь между биномиальными деревьями, из которых состоит H, и бинарным представлением n. Сделайте вывод, что H состоит не более чем из $\lfloor \lg n \rfloor + 1$ биномиальных деревьев.

Предположим, что мы представляем биномиальную пирамиду следующим образом. Каждое биномиальное дерево в биномиальной пирамиде представлено с использованием схемы с левым потомком и правым братом из раздела 10.4. Каждый узел содержит свой ключ, указатель на родительский узел, на крайний слева дочерний узел и на брата, следующего непосредственно справа за ним (эти указатели могут иметь значения NIL в соответствующих ситуациях), а также степень (количество дочерних узлов, как и в фибоначчиевых пирамидах). Корни образуют односвязный список, упорядоченный по степеням корней (в порядке возрастания). Обращение к биномиальной пирамиде выполняется с помощью указателя на первый узел в списке корней.

в. Завершите описание представления биномиальной пирамиды (именуйте атрибуты, опишите, когда они имеют значения NIL, и определите организацию списка корней), а также покажите, как реализовать над биномиальными пирамидами семь операций, которые в разделе были реализованы над фибоначчиевыми пирамидами. Каждая операция должна выполняться в наихудшем случае за время $O(\lg n)$, где n — количество узлов в биномиальной пирамиде (или, в случае операции UNION, в двух объединяемых биномиальных пирамидах). Операция MAKE-HEAP должна выполняться за константное время.

г. Предположим, что нужно реализовать для фибоначчиевой пирамиды только операции над объединяемыми пирамидами (т.е. не нужно реализовывать операции DECREASE-KEY и DELETE). В чем деревья в фибоначчиевой пирамиде схожи с деревьями в биномиальной пирамиде? Чем они различаются? Покажите, что максимальная степень в фибоначчиевой пирамиде с n узлами не превышает $\lfloor \lg n \rfloor$.

д. Профессор разработал новую структуру данных, основанную на фибоначчиевых пирамидах. Его пирамида имеет ту же структуру, что и фибоначчиева пирамида, и поддерживает только операции над объединяемыми пирамидами. Реализации этих операций те же, что и в фибоначчиевых пирамидах, однако вставка и объединение уплотняют список корней в качестве своего последнего

шага. Каковы времена работы операций над пирамидой профессора в наихудшем случае?

19.3. Дополнительные операции над фибоначчиевыми пирамидами

Мы хотим реализовать поддержку двух новых операций над фибоначчиевыми пирамидами, при этом не изменяя амортизированное время работы прочих операций над фибоначчиевыми пирамидами.

а. Операция FIB-HEAP-CHANGE-KEY(H, x, k) изменяет ключ узла x, присваивая ему новое значение k. Приведите эффективную реализацию процедуры FIB-HEAP-CHANGE-KEY и проанализируйте амортизированное время работы вашей реализации для случаев, когда k больше, меньше или равно $x.key$.

б. Разработайте эффективную реализацию процедуры FIB-HEAP-PRUNE(H, r), которая удаляет $q = \min(r, H.n)$ узлов из H. Для удаления могут быть выбраны любые q узлов. Проанализируйте амортизированное время работы вашей реализации. (*Указание:* вам может потребоваться изменение структуры данных и функции потенциала.)

19.4. 2-3-4-пирамиды

В главе 18 рассматривались 2-3-4-деревья, в которых каждый внутренний узел (кроме, возможно, корня) имеет два, три или четыре дочерних узла, и все листья таких деревьев располагаются на одной и той же глубине. В данной задаче мы реализуем ***2-3-4-пирамиды***, поддерживающие операции над объединяемыми пирамидами.

2-3-4-пирамиды отличаются от 2-3-4-деревьев следующим. В 2-3-4-пирамидах ключи хранятся только в листьях, и в каждом листе x хранится ровно один ключ в атрибуте $x.key$. Никакой порядок ключей в листьях не соблюдается, т.е. при перечислении слева направо ключи могут располагаться в произвольном порядке. Каждый внутренний узел x содержит значение $x.small$, которое равно минимальному значению среди ключей листьев поддерева, корнем которого является x. Корень r содержит атрибут $r.height$, в котором хранится высота дерева. И наконец 2-3-4-пирамиды предназначены для хранения в оперативной памяти, так что при работе с ними не требуются никакие операции чтения или записи на диск.

Реализуйте перечисленные далее операции над 2-3-4-пирамидами. Каждая из операций (а)–(д) должна выполняться за время $O(\lg n)$ при работе с 2-3-4-пирамидой, в которой содержится n элементов. Операция UNION в п. (е) должна выполняться за время $O(\lg n)$, где n — количество элементов в двух входных пирамидах.

а. Операция MINIMUM возвращает указатель на лист с наименьшим значением ключа.

б. Операция DECREASE-KEY уменьшает значение ключа в данном листе x до указанного значения $k \leq x.key$.

в. Операция INSERT вставляет в пирамиду лист x с ключом k.

г. Операция DELETE удаляет из пирамиды лист x.

д. Операция EXTRACT-MIN извлекает из пирамиды лист с наименьшим значением ключа.

е. Операция UNION объединяет две 2-3-4-пирамиды, возвращая полученную в результате слияния 2-3-4-пирамиду и уничтожая при этом входные пирамиды.

Заключительные замечания

Фибоначчиевы пирамиды были введены Фредманом (Fredman) и Таржаном (Tarjan) [113]. В их статье описано также приложение фибоначчиевых пирамид к задачам о кратчайших путях из одной вершины, о кратчайших путях между всеми вершинами, о взвешенных паросочетаниях и о минимальном остовном дереве.

Впоследствии Дрисколл (Driscoll), Габов (Gabow), Шрейрман (Sghrairman) и Таржан [95] разработали так называемые "ослабленные пирамиды" (relaxed heaps) в качестве альтернативы фибоначчиевым пирамидам. Они разработали два варианта ослабленных пирамид. Один дает то же амортизированное время работы, что и фибоначчиевы пирамиды, второй же позволяет выполнять операцию DECREASE-KEY в наихудшем случае за (не амортизированное) время $O(1)$, а процедуры EXTRACT-MIN и DELETE в наихудшем случае за время $O(\lg n)$. Ослабленные пирамиды имеют также преимущество перед фибоначчиевыми пирамидами в параллельных алгоритмах.

Обратитесь также к заключительным замечаниям к главе 6, где рассмотрены другие структуры данных, которые поддерживают быстрое выполнение операции DECREASE-KEY, когда последовательность значений, возвращаемых вызовами процедуры EXTRACT-MIN, монотонно растет со временем, а данные представляют собой целые числа в определенном диапазоне.

Глава 20. Деревья ван Эмде Боаса

В предыдущих главах вы познакомились со структурами данных, поддерживающими операции над очередями с приоритетами — бинарными пирамидами в главе 6, красно-черными деревьями в главе 13[1] и фибоначчиевыми пирамидами в главе 19. В каждой из этих структур данных как минимум одна важная операция выполняется за время $O(\lg n)$ — либо в худшем случае, либо в случае амортизированного времени. Фактически, поскольку каждая из указанных структур данных основывается на сравнении ключей, нижняя граница времени сортировки $\Omega(n \lg n)$ из раздела 8.1 говорит о том, что по меньшей мере одна из операций должна выполняться за время $\Omega(\lg n)$. Почему? Если бы мы были способны выполнить и операцию INSERT, и операцию EXTRACT-MIN за время $o(\lg n)$, то мы могли бы отсортировать n ключей за время $o(n \lg n)$, сначала выполнив n операций INSERT, а затем n операций EXTRACT-MIN.

В главе 8 мы, однако, видели, что иногда можно воспользоваться дополнительной информацией о ключах, чтобы отсортировать их за время $o(n \lg n)$. В частности, сортировка подсчетом позволяет отсортировать n ключей, каждый из которых является целым числом в диапазоне от 0 до k, за время $\Theta(n + k)$, которое представляет собой $\Theta(n)$ при $k = O(n)$.

Поскольку нижняя граница сортировки $\Omega(n \lg n)$ может быть превзойдена, когда ключи представляют собой целые числа из ограниченного диапазона, закономерно возникает вопрос — нельзя ли при том же условии выполнять каждую из операций очереди с приоритетами за время $o(\lg n)$? Из данной главы вы узнаете, что это возможно: деревья ван Эмде Боаса поддерживают операции очередей с приоритетами и несколько других операций, со временем работы каждой из них, в наихудшем случае равным $O(\lg \lg n)$. Условием для этого является то, что ключи должны быть целыми числами в диапазоне от 0 до $n - 1$ и не должно быть двух одинаковых ключей.

В частности, деревья ван Эмде Боаса поддерживают все операции над динамическими множествами, перечисленные на с. 261, — SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR. Время работы каждой из них составляет $O(\lg \lg n)$. В данной главе мы опустим рассмотрение сопутству-

---

[1]В главе 13 не обсуждался вопрос реализации операций EXTRACT-MIN и DECREASE-KEY, но их легко построить для любой структуры данных, которая поддерживает операции MINIMUM, DELETE и INSERT.

ющих данных и сосредоточимся только на хранении ключей. Поскольку нас интересуют только ключи, и запрещено наличие одинаковых ключей, вместо операции SEARCH мы реализуем более простую операцию MEMBER(S, x), которая возвращает булево значение, указывающее, имеется ли в настоящий момент в динамическом множестве S значение x.

До сих пор параметр n использовался нами для двух разных целей: как количество элементов в динамическом множестве и как диапазон возможных значений. Во избежание дальнейших недоразумений с этого момента мы будем использовать n для обозначения количества элементов множества в данный момент, а u — для обозначения диапазона возможных значений, так что каждая операция над деревом ван Эмде Боаса выполняется за время $O(\lg\lg u)$. Множество $\{0, 1, 2, \ldots, u-1\}$ будем называть ***универсумом*** (universe) или генеральной совокупностью значений, которые могут храниться в дереве, а u — ***размером универсума***. В этой главе мы полагаем u равным точной степени 2, т.е. $u = 2^k$ для некоторого целого $k \geq 1$.

Раздел 20.1 начинается с рассмотрения некоторых простых подходов, которые укажут нам верное направление. Эти подходы будут усовершенствованы в разделе 20.2 путем введения рекурсивных протоструктур ван Эмде Боаса, которые не позволяют достичь нашей цели — выполнения операций за время $O(\lg\lg u)$. В разделе 20.3 протоструктуры будут модифицированы для создания деревьев ван Эмде Боаса и будет показано, как реализовать каждую операцию так, чтобы она выполнялась за время $O(\lg\lg u)$.

20.1. Предварительные подходы

В этом разделе мы рассмотрим различные подходы к задаче хранения динамического множества. Хотя ни один из них не дает искомое время $O(\lg\lg u)$, мы получим необходимые знания, которые помогут нам понять деревья ван Эмде Боаса, когда позже мы встретимся с ними в этой главе.

Прямая адресация

Прямая адресация, как мы видели в разделе 11.1, предоставляет простейший подход к хранению динамического множества. Поскольку в этой главе нас интересует только хранение ключей, мы можем упростить подход прямой адресации для хранения динамического множества в виде битового вектора, как обсуждалось в упр. 11.1.2. Для хранения динамического множества значений из универсума $\{0, 1, 2, \ldots, u-1\}$ мы поддерживаем массив $A[0\,..\,u-1]$ из u бит. Элемент $A[x]$ хранит 1, если значение x находится в динамическом множестве, и 0 — в противном случае. Хотя каждая из операций INSERT, DELETE и MEMBER при использовании битового вектора выполняется за время $O(1)$, каждая из операций MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR выполняется за время $\Theta(u)$ в наихудшем случае, поскольку нам может потребоваться просканировать $\Theta(u)$

элементов.[2] Например, если множество содержит только значения 0 и $u - 1$, то для поиска последующего за 0 элемента необходимо просканировать элементы с 1 по $u - 2$, перед тем как будет найдено значение 1 в $A[u - 1]$.

Наложение структуры бинарного дерева

Можно сократить длинные сканирования битового вектора, если наложить на него бинарное дерево. Пример показан на рис. 20.1. Элементы битового вектора образуют листья бинарного дерева, и каждый внутренний узел содержит 1 тогда и только тогда, когда некоторый лист в его поддереве содержит 1. Другими словами, бит, хранящийся во внутреннем узле, представляет собой логическое ИЛИ его дочерних узлов.

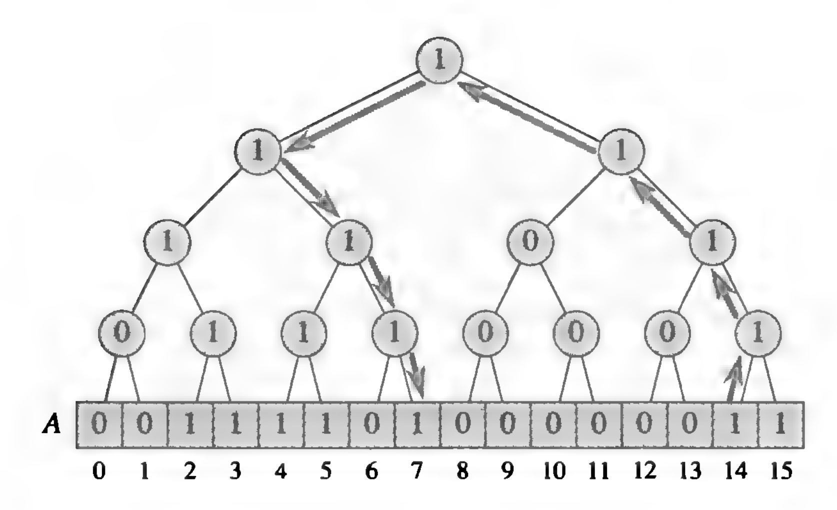


Рис. 20.1. Бинарное дерево битов, наложенное на битовый вектор, представляющий множество $\{2, 3, 4, 5, 7, 14, 15\}$ при $u = 16$. Каждый внутренний узел содержит 1 тогда и только тогда, когда некоторый лист его поддерева содержит 1. Стрелки показывают путь, по которому определяется предшественник 14 в данном множестве.

Операции, которые в простом битовом векторе в наихудшем случае выполнялись за время $\Theta(u)$, теперь используют структуру дерева.

- Для поиска минимального значения множества следует начать работу с корня и двигаться к листьям, всегда выбирая крайний слева узел, содержащий 1.
- Для поиска минимального значения множества следует начать работу с корня и двигаться к листьям, всегда выбирая крайний справа узел, содержащий 1.
- Для поиска преемника x следует начать работу с листа, индексированного x, и двигаться вверх к корню, пока не войдем слева в узел, правый дочерний узел которого z содержит 1. Затем нужно двигаться вниз через узел z, всегда выбирая крайний слева узел, содержащий 1 (т.е. искать минимальное значение в поддереве с корнем в z).

---

[2]В этой главе мы считаем, что процедуры MINIMUM и MAXIMUM в случае пустого множества возвращают NIL, а процедуры SUCCESSOR и PREDECESSOR возвращают это значение, если заданный элемент не имеет соответственно последующего или предшествующего элемента.

- Для поиска предшественника x следует начать работу с листа, индексированного x, и двигаться вверх к корню, пока не войдем справа в узел, левый дочерний узел которого z содержит 1. Затем следует двигаться вниз через узел z, всегда выбирая крайний справа узел, содержащий 1 (т.е. искать максимальное значение в поддереве с корнем в z).

На рис. 20.1 показан путь, пройденный в поисках 7 — предшественника значения 14.

Мы соответствующим образом расширяем также операции INSERT и DELETE. При вставке значения мы сохраняем 1 в каждом узле на простом пути от соответствующего листа вверх до корня. При удалении мы также идем от соответствующего листа до корня, попутно заново вычисляя биты каждого внутреннего узла пути как логическое ИЛИ значений двух его дочерних узлов.

Поскольку высота дерева равна $\lg u$ и каждая из перечисленных операций выполняет не более одного прохода вверх по дереву и не более одного прохода вниз, то в наихудшем случае каждая операция выполняется за время $O(\lg u)$.

Этот подход лишь немногим лучше простого применения красно-черных деревьев. Мы в состоянии выполнить операцию MEMBER за время $O(1)$, в то время как в красно-черном дереве для этого потребовалось бы время $O(\lg n)$. В то же время если количество n элементов множества гораздо меньше размера универсума u, красно-черное дерево окажется быстрее в случае всех прочих операций.

Наложение дерева постоянной высоты

Что произойдет, если выполнить наложение дерева с большей степенью? Пусть размер универсума равен $u = 2^{2k}$ для некоторого целого k, так что $\sqrt{u}$ представляет собой целое число. Вместо наложения поверх битового вектора бинарного дерева выполним наложение дерева со степенью $\sqrt{u}$. На рис. 20.2, (а) показано такое дерево для того же битового вектора, что и на рис. 20.1. Высота такого дерева всегда равна 2.

Как и ранее, каждый внутренний узел хранит результат логического ИЛИ битов в его поддереве, так что $\sqrt{u}$ внутренних узлов на глубине 1 подводят итог для каждой группы из $\sqrt{u}$ значений. Как показано на рис. 20.2, (б), эти узлы можно рассматривать как массив $summary[0\,..\,\sqrt{u}-1]$, где $summary[i]$ содержит 1 тогда

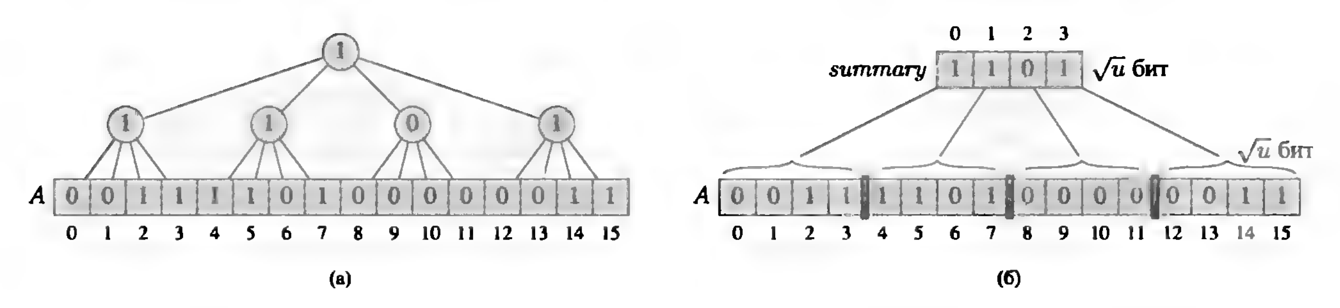


Рис. 20.2. **(а)** Дерево со степенью $\sqrt{u}$, наложенное на тот же битовый вектор, что и на рис. 20.1. Каждый внутренний узел хранит логическое ИЛИ битов своих поддеревьев. **(б)** Вид той же структуры, но с внутренними узлами на глубине 1, рассматриваемыми как массив $summary[0\,..\,\sqrt{u}-1]$, где $summary[i]$ представляет собой логическое ИЛИ подмассива $A[i\sqrt{u}\,..\,(i+1)\sqrt{u}-1]$.

и только тогда, когда подмассив $A[i\sqrt{u} \,..\, (i+1)\sqrt{u} - 1]$ содержит 1. Мы называем этот $\sqrt{u}$-битовый подмассив A i-м ***кластером*** (cluster). Для данного значения x бит $A[x]$ находится в кластере номер $\lfloor x/\sqrt{u} \rfloor$. Теперь процедура INSERT становится операцией со временем работы $O(1)$: для вставки x следует установить $A[x]$ и $summary[\lfloor x/\sqrt{u} \rfloor]$ равными 1. Можно использовать массив $summary$ для выполнения каждой из операций MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR и DELETE за время $O(\sqrt{u})$.

- Для поиска минимального (максимального) значения следует найти крайний слева (справа) элемент $summary$, который содержит 1, скажем, $summary[i]$, а затем выполнить линейный поиск в i-м кластере крайней слева (справа) 1.
- Для поиска преемника (предшественника) x сначала следует выполнить поиск вправо (влево) в пределах его кластера. Если мы находим 1, ее позиция дает нам искомый результат. В противном случае положим $i = \lfloor x/\sqrt{u} \rfloor$ и выполним поиск в массиве $summary$ вправо (влево) от индекса i. Первая позиция, хранящая 1, дает индекс кластера. Поиск в этом кластере крайней слева (справа) 1 дает позицию преемника (предшественника).
- Для удаления значения x положим $i = \lfloor x/\sqrt{u} \rfloor$. Установим $A[x]$ равным 0, а затем установим $summary[i]$ равным логическому ИЛИ битов в i-м кластере.

В каждой из описанных выше операций мы выполняем поиск не более чем в двух кластерах по $\sqrt{u}$ бит, плюс в массиве $summary$, так что каждая операция выполняется за время $O(\sqrt{u})$.

На первый взгляд создается впечатление не прогресса, а регресса. Наложение бинарного дерева дает нам операции со временем выполнения $O(\lg u)$, что асимптотически быстрее, чем время $O(\sqrt{u})$. Однако применение дерева со степенью $\sqrt{u}$ оказывается ключевой идеей деревьев ван Эмде Боаса. Мы продолжим путешествие по этому пути в следующем разделе.

Упражнения

20.1.1

Модифицируйте структуры данных из этого раздела таким образом, чтобы они поддерживали дублирующиеся ключи.

20.1.2

Модифицируйте структуры данных из этого раздела таким образом, чтобы они поддерживали ключи со связанными сопутствующими данными.

20.1.3

Заметьте, что при использовании описанных в этом разделе структур способ поиска преемника и предшественника значения x не зависит от того, находится ли x в данный момент в этом множестве. Покажите, как найти преемник x в бинарном дереве поиска, когда x в этом дереве отсутствует.

20.1.4

Предположим, что вместо дерева со степенью $\sqrt{u}$ мы накладываем дерево со степенью $u^{1/k}$, где $k > 1$ — константа. Какой будет высота такого дерева и за какое время будет выполняться каждая из операций над ним?

20.2. Рекурсивная структура

В этом разделе мы модифицируем идею наложения дерева степени $\sqrt{u}$ на битовый вектор. В предыдущем разделе мы использовали структуру размером $\sqrt{u}$, каждый элемент которой указывал на другую структуру размером $\sqrt{u}$. Теперь мы делаем структуру рекурсивной, уменьшая размер универсума на квадратный корень на каждом уровне рекурсии. Начиная с универсума размером u, мы делаем структуры хранящими $\sqrt{u} = u^{1/2}$ элементов, которые хранят структуры по $u^{1/4}$ элементов, которые хранят структуры по $u^{1/8}$ элементов, и так далее до базового размера 2.

Для простоты в этом разделе мы считаем, что $u = 2^{2^k}$ для некоторого целого числа k, так что $u, u^{1/2}, u^{1/4}, \ldots$ являются целыми числами. На практике это ограничение является достаточно жестким, ограничивающим нас значениями u из последовательности $2, 4, 16, 256, 65536, \ldots$. Из следующего раздела вы узнаете, как ослабить это предположение и считать только, что $u = 2^k$ для некоторого целого числа k. Поскольку рассматриваемая в этом разделе структура представляет собой только предвестник настоящего дерева ван Эмде Боаса, отнесемся терпимо к этому ограничению, призванному способствовать облегчению понимания.

Вспомним, что наша цель — получить время работы операций, равное $O(\lg \lg u)$, и подумаем, как можно ее достичь. В конце раздела 4.3 мы видели, что путем замены переменных можно показать, что рекуррентное соотношение

$$T(n) = 2T\left(\lfloor\sqrt{n}\rfloor\right) + \lg n \tag{20.1}$$

имеет решение $T(n) = O(\lg n \lg \lg n)$. Рассмотрим похожее, но более простое рекуррентное соотношение

$$T(u) = T(\sqrt{u}) + O(1) . \tag{20.2}$$

Если воспользоваться тем же методом замены переменных, можно показать, что рекуррентное соотношение (20.2) имеет решение $T(u) = O(\lg \lg u)$. Пусть $m = \lg u$, так что $u = 2^m$, и имеем

$$T(2^m) = T(2^{m/2}) + O(1) .$$

Теперь переименуем $S(m) = T(2^m)$, что даст новое рекуррентное соотношение

$$S(m) = S(m/2) + O(1) .$$

Согласно случаю 2 основного метода это рекуррентное соотношение имеет решение $S(m) = O(\lg m)$. Вернемся от $S(m)$ к $T(u)$, получая $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg\lg u)$.

Рекуррентное соотношение (20.2) будет направлять наши поиски структуры данных. Мы разработаем рекурсивную структуру данных, которая на каждом уровне рекурсии будет выполнять уменьшение на множитель $\sqrt{u}$. Когда операция обходит эту структуру данных, на каждый уровень, прежде чем опуститься на уровень ниже, она затрачивает константное время. Время выполнения такой операции характеризует рекуррентное соотношение (20.2).

Вот еще один способ представить, как в решении рекуррентного соотношения (20.2) получается $\lg\lg u$. Рассматривая размер универсума на каждом уровне рекурсивной структуры данных, мы получаем последовательность $u, u^{1/2}, u^{1/4}, u^{1/8}, \ldots$. Если рассмотреть, какое количество битов требуется для хранения размера универсума на каждом уровне, мы получим $\lg u$ на верхнем уровне, и каждый последующий уровень будет требовать половину битов предыдущего. В общем случае, если мы начнем с b бит и будем уменьшать это количество на каждом уровне в два раза, то после $\lg b$ уровней мы придем к одному биту. Поскольку $b = \lg u$, после $\lg\lg u$ уровней получаем размер универсума, равный 2.

Вернемся к структуре данных на рис. 20.2. Заданное значение x располагается в кластере с номером $\lfloor x/\sqrt{u} \rfloor$. Если рассматривать x как $\lg u$-битовое бинарное целое число, то номер кластера, $\lfloor x/\sqrt{u} \rfloor$, определяется $(\lg u)/2$ старшими битами числа x. В своем кластере x находится в позиции $x \bmod \sqrt{u}$, которая определяется $(\lg u)/2$ младшими битами числа x. Позже нам потребуется такая индексация, так что определим некоторые функции, которые облегчат нашу работу:

$$\begin{aligned} \text{high}(x) &= \lfloor x/\sqrt{u} \rfloor \ , \\ \text{low}(x) &= x \bmod \sqrt{u} \ , \\ \text{index}(x, y) &= x\sqrt{u} + y \ . \end{aligned}$$

Функция $\text{high}(x)$ дает $(\lg u)/2$ старших битов числа x, возвращая номер кластера x. Функция $\text{low}(x)$ дает $(\lg u)/2$ младших битов числа x и указывает позицию x в его кластере. Функция $\text{index}(x, y)$ строит номер элемента из значений x и y, рассматривая x как $(\lg u)/2$ старших битов номера элемента, а y — как $(\lg u)/2$ младших битов. Справедливо тождество $x = \text{index}(\text{high}(x), \text{low}(x))$. Значение u, используемое каждой из этих функций, всегда представляет собой размер универсума структуры данных, в которой вызывается данная функция и которое изменяется при переходе к рекурсивной структуре.

20.2.1. Протоструктуры ван Эмде Боаса

От рекуррентного соотношения (20.2) перейдем к построению рекурсивной структуры данных, поддерживающей указанные операции. Хотя эта структура данных и не позволяет достичь нашей цели — времени работы $O(\lg\lg u)$ — для некоторых операций, она служит основой для структуры дерева ван Эмде Боаса, с которой мы встретимся в разделе 20.3.

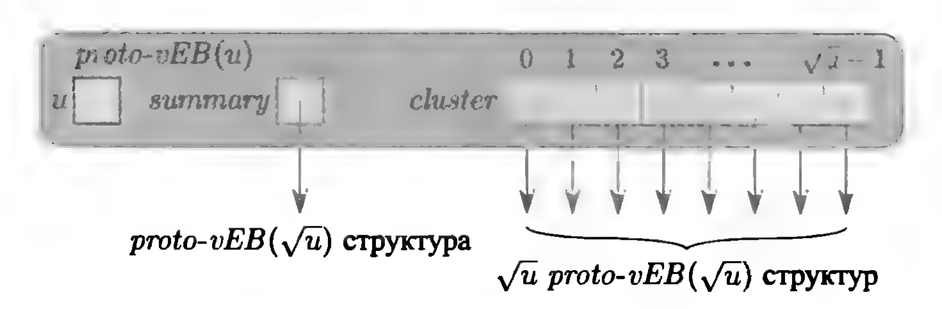


Рис. 20.3. Информация в структуре *proto-vEB*(u) при $u \geq 4$. Структура содержит размер универсума u, указатель *summary* на структуру *proto-vEB*$(\sqrt{u})$ и массив *cluster*$[0\,..\,\sqrt{u}-1]$ из $\sqrt{u}$ указателей на *proto-vEB*$(\sqrt{u})$-структуры.

Рекурсивно определим для универсума $\{0, 1, 2, \ldots, u-1\}$ ***протоструктуру ван Эмде Боаса*** (proto van Emde Boas structure — ***proto-vEB-структуру***), которую обозначим как *proto-vEB*(u). Каждая структура *proto-vEB*(u) содержит атрибут u, указывающий размер универсума. В дополнение к этому структура содержит следующее.

- Если $u = 2$, то это базовый размер, и она содержит массив $A[0\,..\,1]$ из двух битов.
- В противном случае $u = 2^{2^k}$ для некоторого целого числа $k \geq 1$, так что $u \geq 4$. В дополнение к размеру универсума u структура данных *proto-vEB*(u) содержит следующие атрибуты, показанные на рис. 20.3:
  - указатель *summary* на структуру *proto-vEB*$(\sqrt{u})$;
  - массив *cluster*$[0\,..\,\sqrt{u}-1]$ из $\sqrt{u}$ указателей, каждый из которых указывает на *proto-vEB*$(\sqrt{u})$-структуру.

Элемент x, где $0 \leq x < u$, рекурсивно хранится в кластере номер $\text{high}(x)$, как элемент номер $\text{low}(x)$ в пределах этого кластера.

В двухуровневой структуре из предыдущего раздела каждый узел хранит массив размером $\sqrt{u}$, каждый элемент которого содержит один бит. По индексу каждого элемента можно вычислить начальный индекс подмассива размером $\sqrt{u}$, который резюмируется этим битом. В протоструктуре вместо вычисления индексов используются явные указатели. Массив *summary* содержит резюмирующие биты, рекурсивно хранящиеся в протоструктуре, и массив *cluster*, содержащий $\sqrt{u}$ указателей.

На рис. 20.4 показана (полностью развернутая) протоструктура *proto-vEB*(16), представляющая множество $\{2, 3, 4, 5, 7, 14, 15\}$. Если значение i находится в протоструктуре, на которую указывает *summary*, то i-й кластер содержит некоторое значение из представленного множества. Как и в дереве с константной высотой, *cluster*$[i]$ представляет значения от $i\sqrt{u}$ до $(i+1)\sqrt{u}-1$, которые образуют i-й кластер.

На базовом уровне элементы фактических динамических множеств хранятся в некоторых из структур *proto-vEB*(2), а остальные структуры *proto-vEB*(2) хранят резюмирующие биты. На рисунке под каждой из нерезюмирующих базовых структур указано, какие биты в ней хранятся. Например, структура *proto-vEB*(2),

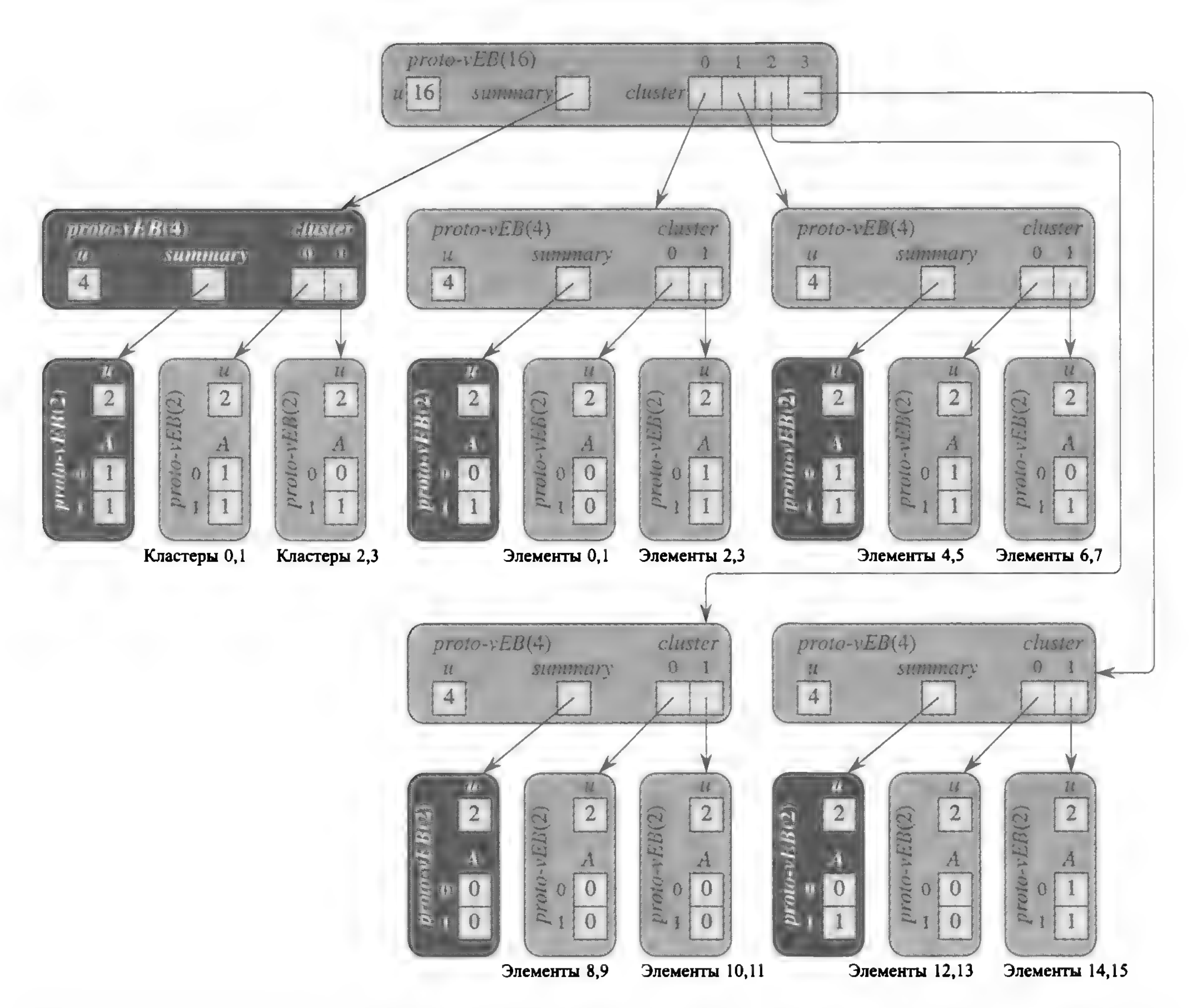


Рис. 20.4. Структура *proto-vEB*(16), представляющая множество $\{2, 3, 4, 5, 7, 14, 15\}$. Она указывает на четыре структуры *proto-vEB*(4) в *cluster*$[0..3]$ и на резюмирующую структуру, которая также представляет собой *proto-vEB*(4). Каждая структура *proto-vEB*(4) указывает на две структуры *proto-vEB*(2) в *cluster*$[0..1]$ и на резюмирующую *proto-vEB*(2). Каждая структура *proto-vEB*(2) содержит только массив $A[0..1]$ из двух битов. Структуры *proto-vEB*(2) над "Элементы *i,j*" хранят биты i и j фактического динамического множества, а структуры *proto-vEB*(2) над "Кластеры *i,j*" хранят биты *summary* для кластеров i и j в структуре *proto-vEB*(16) верхнего уровня. Для ясности темная штриховка указывает протоструктуру верхнего уровня, которая хранит резюмирующую информацию для родительской структуры; во всем остальном такая протоструктура идентична другим протоструктурам с тем же размером универсума.

помеченная как "Элементы 6,7", хранит бит 6 (нулевой, поскольку элемент 6 в множестве отсутствует) в своем элементе $A[0]$ и бит 7 (единичный, поскольку элемент 7 присутствует в множестве) в своем элементе $A[1]$.

Подобно кластерам, каждая резюмирующая структура представляет собой просто динамическое множество с размером универсума $\sqrt{u}$, так что мы представляем ее с помощью структуры *proto-vEB*$(\sqrt{u})$. Четыре резюмирующих бита для основной структуры *proto-vEB*(16) находятся в крайней слева структуре *proto-vEB*(4) и в конечном итоге — в двух структурах *proto-vEB*(2). Например, структура *proto-vEB*(2), помеченная как "Кластеры 2,3", имеет $A[0] = 0$, что говорит о том, что в кластере 2 структуры *proto-vEB*(16) (содержащем элементы

8–11) содержатся только нули 0, и $A[1] = 1$, что говорит нам о том, что в кластере 3 (содержащем элементы 12–15) имеется как минимум одна единица. Каждая структура *proto-vEB*(4) указывает на свое резюме, хранящееся в виде структуры *proto-vEB*(2). Взгляните, например, на структуру *proto-vEB*(2) слева от структуры, помеченной как "Элементы 0,1". Поскольку ее элемент $A[0]$ равен 0, значит, все элементы структуры "Элементы 0,1" нулевые, а поскольку ее элемент $A[1]$ равен 1, мы знаем, что структура "Элементы 2,3" содержит как минимум одну единицу.

20.2.2. Операции с протоструктурами ван Эмде Боаса

Теперь опишем, как выполняются операции над протоструктурами ван Эмде Боаса. Сначала будут рассмотрены запрашивающие операции MEMBER, MINIMUM и SUCCESSOR, которые не изменяют протоструктуру. Затем рассмотрим операции INSERT и DELETE. Операции MAXIMUM и PREDECESSOR, симметричные операциям MINIMUM и SUCCESSOR соответственно, мы оставим в качестве упр. 20.2.1.

Каждая из операций MEMBER, SUCCESSOR, PREDECESSOR, INSERT и DELETE получает параметр x и протоструктуру V. В каждой операции предполагается, что $0 \leq x < V.u$.

Определение наличия значения в множестве

Для выполнения операции MEMBER(x) необходимо найти бит для x в соответствующей структуре *proto-vEB*(2). Это можно сделать за время $O(\lg \lg u)$, полностью минуя структуры *summary*. Приведенная далее процедура получает *proto-vEB*-структуру V и значение x, и возвращает бит, указывающий, входит ли x в динамическое множество, хранимое структурой V.

```
PROTO-VEB-MEMBER(V, x)
1  if V.u == 2
2      return V.A[x]
3  else return PROTO-VEB-MEMBER(V.cluster[high(x)], low(x))
```

Процедура PROTO-VEB-MEMBER работает следующим образом. В строке 1 выполняется проверка базового случая, когда V представляет собой структуру *proto-vEB*(2). Строка 2 обрабатывает базовый случай, просто возвращая соответствующий бит массива A. Строка 3 работает рекурсивно, "погружаясь" в соответствующую меньшую протоструктуру. Значение high(x) указывает, какая структура *proto-vEB*($\sqrt{u}$) посещается, а low(x) определяет запрашиваемый в структуре *proto-vEB*($\sqrt{u}$) элемент.

Давайте посмотрим, что происходит при вызове PROTO-VEB-MEMBER(V, 6) для структуры *proto-vEB*(16) на рис. 20.4. Поскольку high(6) = 1 при $u = 16$, мы рекурсивно обращаемся к структуре *proto-vEB*(4) справа вверху и запрашиваем элемент low(6) = 2 этой структуры. В этом рекурсивном вызове $u = 4$, так что рекурсия продолжается. В случае $u = 4$ мы имеем high(2) = 1 и low(2) = 0,

19 Зак. 3726

так что мы запрашиваем элемент 0 структуры *proto-vEB*(2) справа вверху. Этот рекурсивный вызов оказывается базовым случаем, так что цепочка рекурсивных вызовов возвращает $A[0] = 0$. Таким образом, вызов PROTO-VEB-MEMBER$(V, 6)$ возвращает 0, указывая, что элемент 6 в множестве отсутствует.

Для определения времени работы PROTO-VEB-MEMBER обозначим время работы со структурой *proto-vEB*(u) через $T(u)$. Каждый рекурсивный вызов требует константного времени (не включая время, затрачиваемое вызовами, которые он выполняет в свою очередь). Когда процедура PROTO-VEB-MEMBER делает рекурсивный вызов, она обращается к структуре *proto-vEB*$(\sqrt{u})$. Таким образом, время работы можно характеризовать рекуррентным соотношением $T(u) = T(\sqrt{u}) + O(1)$, с которым мы уже встречались в (20.2). Его решение имеет вид $T(u) = O(\lg\lg u)$, и мы можем заключить, что время работы процедуры PROTO-VEB-MEMBER составляет $O(\lg\lg u)$.

Поиск минимального элемента

Теперь рассмотрим, как выполнить операцию MINIMUM. Процедура PROTO-VEB-MINIMUM(V) возвращает минимальный элемент в протоструктуре V, или NIL, если V представляет пустое множество.

```
PROTO-VEB-MINIMUM(V)
 1  if V.u == 2
 2      if V.A[0] == 1
 3          return 0
 4      elseif V.A[1] == 1
 5          return 1
 6      else return NIL
 7  else min-cluster = PROTO-VEB-MINIMUM(V.summary)
 8      if min-cluster == NIL
 9          return NIL
10      else offset = PROTO-VEB-MINIMUM(V.cluster[min-cluster])
11          return index(min-cluster, offset)
```

Эта процедура работает следующим образом. В строке 1 выполняется проверка базового случая, обрабатываемого в строках 2–6 методом "грубой силы". Рекурсивный случай обрабатывается строками 7–11. Сначала в строке 7 определяется номер первого кластера, который содержит элемент множества. Это делается с помощью применения рекурсивного вызова PROTO-VEB-MINIMUM к атрибуту *V.summary*, который представляет собой структуру *proto-vEB*$(\sqrt{u})$. В строке 7 этот номер кластера присваивается переменной *min-cluster*. Если множество пустое, то рекурсивный вызов вернет значение NIL, которое будет возвращено рассматриваемой процедурой в строке 9. В противном случае минимальный элемент множества находится где-то в кластере с номером *min-cluster*. Рекурсивный вызов в строке 10 находит смещение минимального элемента в пределах этого кластера. Наконец в строке 11 из номера кластера и смещения в кластере строится минимальный элемент и возвращается его значение.

Хотя запрос резюмирующей информации позволяет быстро найти кластер, содержащий минимальный элемент (поскольку эта процедура выполняет два рекурсивных вызова над структурами *proto-vEB*$(\sqrt{u})$), в наихудшем случае время ее работы не равно $O(\lg\lg u)$. Обозначив через $T(u)$ время работы процедуры PROTO-VEB-MINIMUM со структурой *proto-vEB*(u) в наихудшем случае, получаем рекуррентное соотношение

$$T(u) = 2T(\sqrt{u}) + O(1) . \tag{20.3}$$

Вновь применим замену переменных для решения этого рекуррентного соотношения, положив $m = \lg u$. Это дает

$$T(2^m) = 2T(2^{m/2}) + O(1) .$$

Переименование $S(m) = T(2^m)$ дает рекуррентное соотношение

$$S(m) = 2S(m/2) + O(1) ,$$

которое, согласно случаю 1 основной теоремы, имеет решение $S(m) = \Theta(m)$. Возвращаясь от $S(m)$ к $T(u)$, получим $T(u) = T(2^m) = S(m) = \Theta(m) = \Theta(\lg u)$. Таким образом, мы видим, что из-за второго рекурсивного вызова время работы процедуры PROTO-VEB-MINIMUM равно $\Theta(\lg u)$, а не требуемому $O(\lg\lg u)$.

Поиск преемника

Операция SUCCESSOR еще хуже. В наихудшем случае наряду с вызовом PROTO-VEB-MINIMUM она делает еще два рекурсивных вызова. Процедура PROTO-VEB-SUCCESSOR(V, x) возвращает наименьший элемент в протоструктуре V, больший, чем x, или значение NIL, если в V нет элемента, большего, чем x. Условие, чтобы элемент x имелся в множестве, не ставится, но предполагается, что $0 \leq x < V.u$.

```
PROTO-VEB-SUCCESSOR(V, x)
if V.u == 2
    if x == 0 и V.A[1] == 1
        return 1
    else return NIL
else offset = PROTO-VEB-SUCCESSOR(V.cluster[high(x)], low(x))
    if offset ≠ NIL
        return index(high(x), offset)
    else succ-cluster = PROTO-VEB-SUCCESSOR(V.summary, high(x))
        if succ-cluster == NIL
            return NIL
        else offset = PROTO-VEB-MINIMUM(V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

Процедура Proto-vEB-Successor работает следующим образом. Как обычно, в строке 1 проверяется базовый случай, обрабатываемый методом грубой силы в строках 2–4: единственный вариант наличия преемника у x в структуре *proto-vEB*(2) — при $x = 0$ и $A[1]$, равном 1. В строках 5–12 выполняется обработка рекурсивного случая. В строке 5 выполняется поиск преемника x в кластере x с присваиванием результата переменной *offset*. В строке 6 определяется, имеет ли x преемника в своем кластере; если имеет, то строка 7 вычисляет и возвращает значение этого преемника. В противном случае требуется выполнить поиск в других кластерах. В строке 8 переменной *succ-cluster* присваивается номер следующего непустого кластера с использованием резюмирующей информации для его поиска. В строке 9 выполняется проверка равенства *succ-cluster* значению NIL, при этом, если все последующие кластеры пусты, в строке 10 процедуры возвращается значение NIL. Если *succ-cluster* не равно NIL, то в строке 11 переменной *offset* присваивается первый элемент этого кластера, а строка 12 вычисляет и возвращает минимальный элемент в данном кластере.

В наихудшем случае процедура Proto-vEB-Successor рекурсивно вызывает сама себя дважды для структур *proto-vEB*$(\sqrt{u})$ и делает один вызов Proto-vEB-Minimum для структуры *proto-vEB*$(\sqrt{u})$. Таким образом, рекуррентное соотношение для времени работы процедуры Proto-vEB-Successor в наихудшем случае $T(u)$ имеет вид

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + \Theta(\lg \sqrt{u}) \\ &= 2T(\sqrt{u}) + \Theta(\lg u) \;. \end{aligned}$$

Можно применить ту же методику, что и для рекуррентного соотношения (20.1), и показать, что данное рекуррентное соотношение имеет решение $T(u) = \Theta(\lg u \lg\lg u)$. Таким образом, Proto-vEB-Successor асимптотически медленнее Proto-vEB-Minimum.

Вставка элемента

Для вставки элемента необходимо вставить его в соответствующий кластер, а также установить резюмирующий бит этого кластера равным 1. Процедура Proto-vEB-Insert(V, x) вставляет значение x в протоструктуру V.

```
Proto-vEB-Insert(V, x)
1  if V.u == 2
2      V.A[x] = 1
3  else Proto-vEB-Insert(V.cluster[high(x)], low(x))
4      Proto-vEB-Insert(V.summary, high(x))
```

В базовом случае строка 2 устанавливает соответствующий бит в массиве A равным 1. В рекурсивном случае рекурсивный вызов в строке 3 вставляет x в соответствующий кластер, а строка 4 устанавливает резюмирующий бит этого кластера равным 1.

Поскольку процедура PROTO-vEB-INSERT в наихудшем случае делает два рекурсивных вызова, ее время работы характеризуется рекуррентным соотношением (20.3). Следовательно, время работы процедуры PROTO-vEB-INSERT равно $\Theta(\lg u)$.

Удаление элемента

Операция DELETE более сложная, чем операция вставки. В то время как при вставке всегда можно установить резюмирующий бит равным 1, при удалении сбросить резюмирующий бит в 0 можно не всегда. Нужно определить, есть ли в данном кластере единичные биты. Исходя из определения протоструктур, требуется исследовать все $\sqrt{u}$ бит кластера, чтобы выяснить, есть ли среди них хотя бы один, равный 1. В качестве альтернативы в протоструктуру можно добавить атрибут n, указывающий количество элементов в ней. Реализация процедуры PROTO-vEB-DELETE оставлена читателям в качестве упр. 20.2.2 и 20.2.3.

Очевидно, что нам нужно так модифицировать протоструктуру ван Эмде Боаса таким образом, чтобы каждая операция выполняла не более одного рекурсивного вызова. Из следующего раздела вы узнаете, как это можно сделать.

Упражнения

20.2.1

Напишите псевдокоды процедур PROTO-vEB-MAXIMUM и PROTO-vEB-PREDECESSOR.

20.2.2

Напишите псевдокод процедуры PROTO-vEB-DELETE. Она должна обновлять соответствующий резюмирующий бит путем сканирования соответствующих битов кластера. Каково время работы вашей процедуры в наихудшем случае?

20.2.3

Добавьте к каждой протоструктуре атрибут n, указывающий количество элементов, которое она представляет во множестве в данный момент, и напишите псевдокод процедуры PROTO-vEB-DELETE, использующий этот атрибут n для принятия решения о сбросе резюмирующих битов в 0. Каково время работы вашей процедуры в наихудшем случае? Какие другие процедуры должны быть изменены в связи с добавлением этого нового атрибута? Повлияют ли эти изменения на времена работы этих процедур?

20.2.4

Модифицируйте протоструктуру таким образом, чтобы она поддерживала дублирующиеся ключи.

20.2.5

Модифицируйте протоструктуру таким образом, чтобы она поддерживала ключи со связанными с ними сопутствующими данными.

20.2.6

Напишите псевдокод процедуры, создающей протоструктуру *proto-vEB*(u).

20.2.7

Докажите, что если выполняется строка 9 процедуры PROTO-VEB-MINIMUM, то протоструктура пуста.

20.2.8

Предположим, что разработана протоструктура, в которой каждый массив *cluster* имеет только $u^{1/4}$ элементов. Какими будут времена работы каждой из операций?

20.3. Дерево ван Эмде Боаса

Протоструктура из предыдущего раздела не позволяет достичь искомого времени работы $O(\lg\lg u)$. Это связано с тем, что в большинстве операций требуется слишком много рекурсивных вызовов. В этом разделе мы разработаем структуру данных, подобную протоструктуре ван Эмде Боаса, но хранящую немного больше информации и тем самым устраняющую необходимость в части рекурсии.

В разделе 20.2 мы видели, что сделанное предположение о размере универсума — $u = 2^{2^k}$ для некоторого целого k — неоправданно ограничивает возможные значения u слишком разреженным множеством значений. Начиная с этого момента мы позволяем размеру универсума быть любой точной степенью 2 и, когда $\sqrt{u}$ не является целым числом (т.е. u представляет собой нечетную степень 2: $u = 2^{2k+1}$ для некоторого целого $k \geq 0$), мы будем делить $\lg u$ бит числа на старшие $\lceil(\lg u)/2\rceil$ бит и младшие $\lfloor(\lg u)/2\rfloor$ бит. Для удобства обозначим $2^{\lceil(\lg u)/2\rceil}$ ("верхний квадратный корень" u) как $\sqrt[\uparrow]{u}$, а $2^{\lfloor(\lg u)/2\rfloor}$ ("нижний квадратный корень" u) как $\sqrt[\downarrow]{u}$, так что $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$ и, когда u представляет собой четную степень 2 ($u = 2^{2k}$ для некоторого целого k), $\sqrt[\uparrow]{u} = \sqrt[\downarrow]{u} = \sqrt{u}$. Поскольку мы позволяем u быть нечетной степенью 2, следует переопределить вспомогательные функции из раздела 20.2:

$$\begin{aligned} \text{high}(x) &= \lfloor x/\sqrt[\downarrow]{u} \rfloor \ , \\ \text{low}(x) &= x \bmod \sqrt[\downarrow]{u} \ , \\ \text{index}(x,y) &= x\sqrt[\downarrow]{u} + y \ . \end{aligned}$$

20.3.1. Деревья ван Эмде Боаса

Дерево ван Эмде Боаса (van Emde Boas tree — vEB-дерево), модифицирует протоструктуру ван Эмде Боаса. Обозначим vEB-дерево с размером универсума u как $vEB(u)$ и, если только u не равно базовому размеру 2, с атрибутом *summary*, указывающим на дерево $vEB(\sqrt[\uparrow]{u})$, и с массивом *cluster*$[0\,..\,\sqrt[\uparrow]{u}-1]$, указывающим на деревья $\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$. Как показано на рис. 20.5, vEB-дерево содержит два атрибута, отсутствующие в протоструктуре:

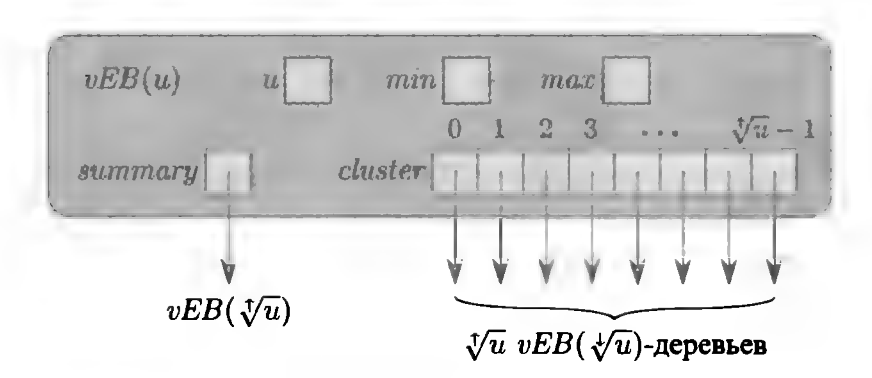


Рис. 20.5. Информация в $vEB(u)$-дереве при $u > 2$. Структура содержит размер универсума u, элементы min и max, указатель $summary$ на дерево $vEB(\sqrt[\uparrow]{u})$ и массив $cluster[0\,..\,\sqrt[\uparrow]{u}-1]$ из $\sqrt[\uparrow]{u}$ указателей на $vEB(\sqrt[\downarrow]{u})$-деревья.

- min хранит минимальный элемент vEB-дерева;
- max хранит максимальный элемент vEB-дерева.

Кроме того, элемент, хранящийся в min, не появляется ни в одном из рекурсивных деревьев $vEB(\sqrt[\downarrow]{u})$, на которые указывает массив $cluster$. Таким образом, элементы, хранящиеся в $vEB(u)$-дереве V, представляют собой $V.min$ плюс все элементы, рекурсивно хранящиеся в $vEB(\sqrt[\downarrow]{u})$-деревьях, на которые указывают $V.cluster[0\,..\,\sqrt[\uparrow]{u}-1]$. Заметим, что, когда vEB-дерево содержит два или более элементов, мы рассматриваем min и max по-разному: элемент, хранящийся в min, не появляется ни в одном из кластеров, в отличие от элемента max, который располагается в кластере (кроме случая, когда vEB-дерево содержит единственный элемент, так что минимальный и максимальный элементы совпадают).

Поскольку базовый размер равен 2, $vEB(2)$-дерево не нуждается в массиве A, который имеет соответствующая структура $proto\text{-}vEB(2)$. Вместо этого его элементы можно определить из атрибутов min и max. В vEB-дереве без элементов, независимо от его размера универсума u, оба атрибута, min и max, равны NIL.

На рис. 20.6 показано $vEB(16)$-дерево V, хранящее массив $\{2, 3, 4, 5, 7, 14, 15\}$. Поскольку наименьшим элементом является 2, атрибут $V.min$ равен 2, и несмотря на то что $\text{high}(2) = 0$, этот элемент 2 отсутствует в $vEB(4)$-дереве, на которое указывает $V.cluster[0]$: обратите внимание, что $V.cluster[0].min$ равно 3, так что 2 в этом vEB-дереве отсутствует. Аналогично, поскольку $V.cluster[0].min$ равно 3, и 2, и 3 являются единственными элементами в $V.cluster[0]$, кластеры $vEB(2)$ в $V.cluster[0]$ пустые.

Атрибуты min и max являются ключевыми элементами для снижения количества рекурсивных вызовов в операциях над vEB-деревьями. Эти атрибуты служат четырем целям.

1. Операции MINIMUM и MAXIMUM не выполняют ни одного рекурсивного вызова, просто возвращая значения min или max.
2. Операция SUCCESSOR может избежать рекурсивных вызовов для определения, находится ли преемник значения x в $\text{high}(x)$. Это связано с тем, что преемник x находится в его кластере тогда и только тогда, когда x строго меньше атрибута max этого кластера. Симметричные рассуждения применимы к операции PREDECESSOR и атрибуту min.

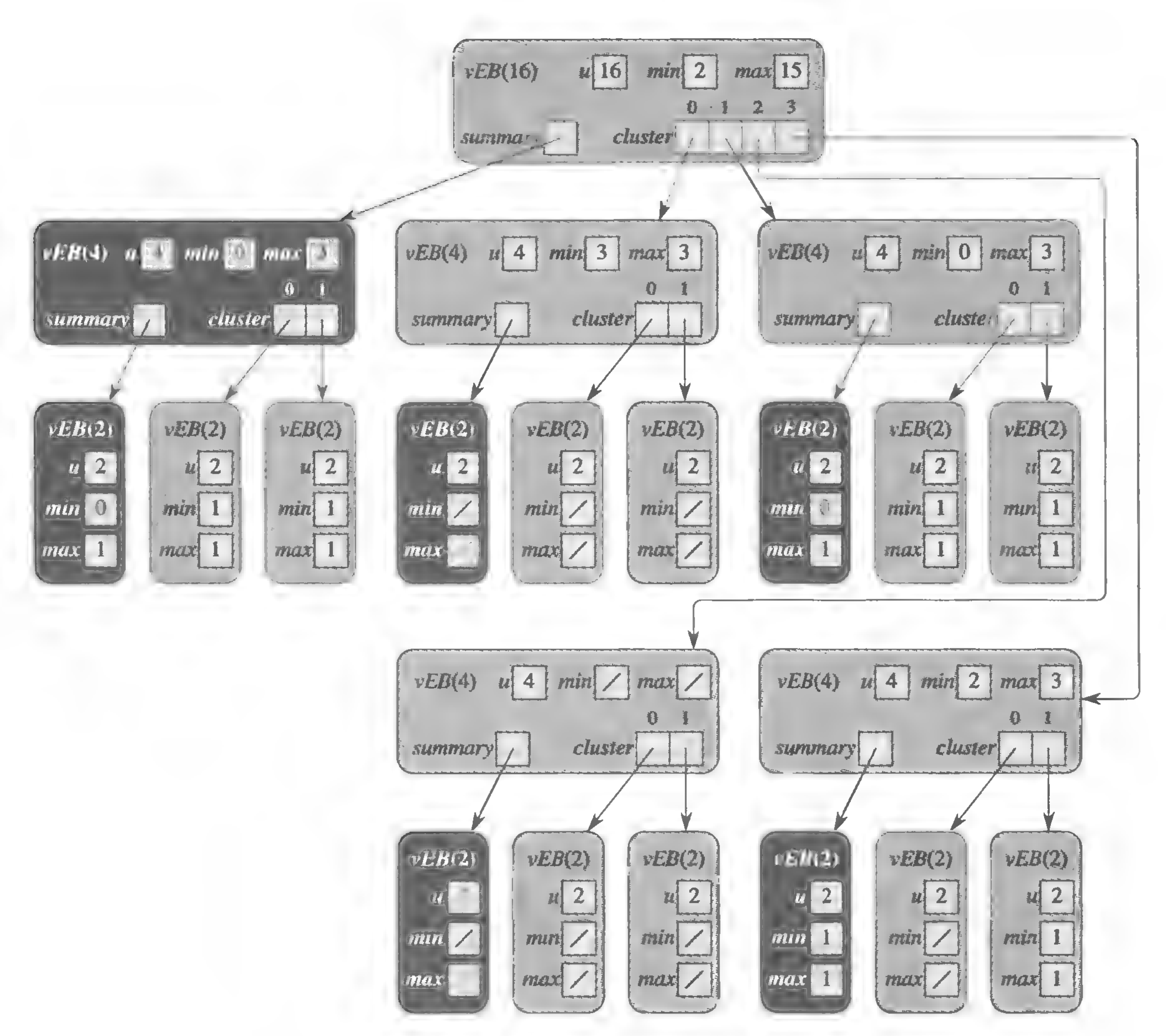


Рис. 20.6. $vEB(16)$-дерево, соответствующее *proto-vEB*-дереву на рис. 20.4. Оно хранит множество $\{2, 3, 4, 5, 7, 14, 15\}$. Косые черты указывают значения NIL. Значение, хранящееся в атрибуте min vEB-дерева, не появляется ни в одном из его кластеров. Темная штриховка служит той же цели, что и на рис. 20.4.

3. Исходя из значений атрибутов min и max vEB-дерева, можно за константное время определить, пустое ли оно или в нем есть один или как минимум два элемента. Эта возможность используется в операциях INSERT и DELETE. Если и min, и max равны NIL, то vEB-дерево не содержит ни одного элемента. Если min и max не равны NIL, но равны друг другу, значит, в vEB-дереве только один элемент. В противном случае и min, и max не равны ни NIL, ни друг другу, и vEB-дерево содержит два или более элементов.

4. Зная, что vEB-дерево пустое, мы можем вставить в него элемент, просто обновляя его атрибуты min и max. Следовательно, вставку в пустое дерево можно выполнить за константное время. Аналогично, если мы знаем, что vEB-дерево содержит только один элемент, его можно удалить за константное время простым обновлением атрибутов min и max. Эти свойства позволят сократить цепочки рекурсивных вызовов.

Даже если размер универсума u представляет собой нечетную степень 2, разница в размерах резюмирующего vEB-дерева и кластеров не повлияет на асимптотическое время работы операций над vEB-деревом. Все рекурсивные процедуры, реализующие операции над vEB-деревом, характеризуются рекуррентным соотношением

$$T(u) \le T(\sqrt[\uparrow]{u}) + O(1) . \qquad (20.4)$$

Это рекуррентное соотношение похоже на рекуррентное соотношение (20.2), и решать его мы будем таким же способом. Положив $m = \lg u$, перепишем рекуррентное соотношение в виде

$$T(2^m) \le T(2^{\lceil m/2 \rceil}) + O(1) .$$

Заметив, что $\lceil m/2 \rceil \le 2m/3$ для всех $m \ge 2$, мы имеем

$$T(2^m) \le T(2^{2m/3}) + O(1) .$$

Полагая $S(m) = T(2^m)$, перепишем это последнее рекуррентное соотношение как

$$S(m) \le S(2m/3) + O(1) ,$$

которое, в соответствии со случаем 2 основного метода, имеет решение $S(m) = O(\lg m)$. (В терминах асимптотического решения дробь $2/3$ ничем не отличается от дроби $1/2$, поскольку при применении основного метода мы находим, что $\log_{3/2} 1 = \log_2 1 = 0$.) Таким образом, мы имеем $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

Прежде чем использовать дерево ван Эмде Боаса, необходимо знать размер универсума u, чтобы иметь возможность создать представляющее пустое множество дерево ван Эмде Боаса соответствующего размера. В задаче 20.1 требуется показать, что общее количество памяти, требующееся для дерева ван Эмде Боаса, равно $O(u)$, а время создания пустого дерева составляет $\Theta(u)$. Вспомним, что, напротив, пустое красно-черное дерево можно создать за константное время. Таким образом, мы можем отказаться от использования дерева ван Эмде Боаса в случае небольшого количества операций, поскольку время, требующееся для создания структуры данных, может превзойти экономию времени на отдельных операциях. Обычно этот недостаток не является существенным, поскольку, как правило, для представления множества с малым количеством элементов используются простые структуры данных, такие как массив или связанный список.

20.3.2. Операции над деревом ван Эмде Боаса

Теперь мы готовы рассмотреть выполнение операций над деревом ван Эмде Боаса. Как и в случае протоструктуры ван Эмде Боаса, рассмотрим сначала запрашивающие операции, а затем — операции INSERT и DELETE. Из-за небольшой асимметрии между минимальным и максимальным элементами в vEB-дереве — когда оно содержит как минимум два элемента, минимальный элемент, в отличие от максимального, в кластере не содержится — мы приведем псевдокоды для всех

пяти запрашивающих операций. Как и в случае операций над протоструктурами ван Эмде Боаса, рассматриваемые здесь операции получают в качестве параметров V и x, где V представляет собой дерево ван Эмде Боаса, а x — элемент; при этом предполагается, что $0 \leq x < V.u$.

Поиск минимального и максимального элементов

Поскольку минимальный и максимальный элементы хранятся в виде атрибутов *min* и *max*, эти две операции имеют однострочный код и выполняются за константное время.

```
VEB-TREE-MINIMUM(V)
1  return V.min
```

```
VEB-TREE-MAXIMUM(V)
1  return V.max
```

Определение наличия значения в множестве

Процедура VEB-TREE-MEMBER(V, x) имеет рекурсивный случай наподобие процедуры PROTO-VEB-MEMBER, но базовые случаи несколько отличаются. Мы также проверяем, не равен ли x минимальному или максимальному элементу. Поскольку vEB-дерево не хранит биты, как это делает протоструктура, процедура VEB-TREE-MEMBER разработана таким образом, чтобы возвращать значения TRUE или FALSE, а не 1 или 0.

```
VEB-TREE-MEMBER(V, x)
1  if x == V.min или x == V.max
2      return TRUE
3  elseif V.u == 2
4      return FALSE
5  else return VEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

В строке 1 выполняется проверка, не является ли x минимальным или максимальным элементом. Если является, то строка 2 возвращает значение TRUE. В противном случае в строке 3 выполняется проверка базового случая. Поскольку в базовом случае дерево $vEB(2)$ не имеет элементов, отличных от хранящихся в атрибутах *min* и *max*, строка 4 возвращает значение FALSE. Другая возможность — когда случай не базовый, а x не равно ни *min*, ни *max* — обрабатывается рекурсивным вызовом в строке 5.

Время работы процедуры VEB-TREE-MEMBER характеризует рекуррентное соотношение (20.4), так что эта процедура выполняется за время $O(\lg \lg u)$.

Поиск преемника и предшественника

Теперь рассмотрим, как реализовать операцию SUCCESSOR. Вспомним, что процедура PROTO-VEB-SUCCESSOR(V, x) может выполнять два рекурсивных вызова: один для определения того, находится ли преемник x в том же кластере, что

и x, и если нет, то еще один для поиска кластера, содержащего преемник x. Поскольку в дереве ван Эмде Боаса можно быстро получить доступ к максимальному элементу, двух рекурсивных вызовов можно избежать, ограничившись вместо них одним рекурсивным вызовом либо над кластером, либо над резюмирующим массивом, но не над обоими.

```
vEB-TREE-SUCCESSOR(V, x)
 1  if V.u == 2
 2      if x == 0 и V.max == 1
 3          return 1
 4      else return NIL
 5  elseif V.min ≠ NIL и x < V.min
 6      return V.min
 7  else max-low = vEB-TREE-MAXIMUM(V.cluster[high(x)])
 8      if max-low ≠ NIL и low(x) < max-low
 9          offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
12          if succ-cluster == NIL
13              return NIL
14          else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
15              return index(succ-cluster, offset)
```

В этой процедуре шесть инструкций **return** и несколько ветвлений. Мы начинаем с базового случая в строках 2–4, возвращая при этом 1 в строке 3, если выполняется попытка найти преемник 0 и 1 находится в 2-элементном множестве; в противном случае базовый случай возвращает NIL в строке 4.

Если мы имеем дело не с базовым случаем, то следующей в строке 5 выполняется проверка, не является ли x строго меньше минимального элемента. Если это так, мы просто возвращаем минимальный элемент в строке 6.

Если мы добрались до строки 7, то нам известно, что мы имеем дело не с базовым случаем и что x не меньше минимального значения vEB-дерева V. В строке 7 выполняется присваивание переменной *max-low* максимального элемента кластера x. Если кластер x содержит несколько элементов, бóльших x, то нам известно, что преемник x находится где-то в кластере x. В строке 8 выполняется проверка этого условия. Если преемник x находится в кластере x, то в строке 9 выясняется, где именно в кластере он находится, а строка 10 возвращает преемник точно так же, как и строка 7 процедуры PROTO-vEB-SUCCESSOR.

Мы попадаем в строку 11, если x не меньше, чем наибольший элемент в его кластере. В этом случае в строках 11–15 поиск преемника x выполняется таким же образом, как и в строках 8–12 процедуры PROTO-vEB-SUCCESSOR.

Легко увидеть, как рекуррентное соотношение (20.4) характеризует время работы процедуры vEB-TREE-SUCCESSOR. В зависимости от результата тестирования в строке 8 процедура рекурсивно вызывает сама себя либо в строке 9 (для vEB-дерева с размером универсума $\sqrt[\downarrow]{u}$), либо в строке 11 (для vEB-дерева с раз-

мером универсума $\sqrt[4]{u}$). В любом случае выполняется один рекурсивный вызов для vEB-дерева с размером универсума не более $\sqrt[4]{u}$. Оставшаяся часть процедуры, включая вызовы vEB-TREE-MINIMUM и vEB-TREE-MAXIMUM, выполняется за время $O(1)$. Следовательно, в наихудшем случае время работы процедуры vEB-TREE-SUCCESSOR составляет $O(\lg \lg u)$.

Процедура vEB-TREE-PREDECESSOR симметрична процедуре vEB-TREE-SUCCESSOR, но с одним дополнительным ветвлением.

```
vEB-TREE-PREDECESSOR(V, x)
 1  if V.u == 2
 2      if x == 1 и V.min == 0
 3          return 0
 4      else return NIL
 5  elseif V.max ≠ NIL и x > V.max
 6      return V.max
 7  else min-low = vEB-TREE-MINIMUM(V.cluster[high(x)])
 8      if min-low ≠ NIL и low(x) > min-low
 9          offset = vEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
12          if pred-cluster == NIL
13              if V.min ≠ NIL и x > V.min
14                  return V.min
15              else return NIL
16          else offset = vEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17              return index(pred-cluster, offset)
```

В строках 13 и 14 образуется дополнительное ветвление. Этот случай осуществляется, когда предшественник x, если таковой существует, располагается не в кластере x. В процедуре vEB-TREE-SUCCESSOR мы точно знали, что если преемник x находится вне кластера x, то он должен находиться в кластере с бо́льшим номером. Но если предшественник x является минимальным элементом vEB-дерева V, то этот предшественник вообще не находится в кластере. В строке 13 выполняется проверка этого условия, а в строке 14 при необходимости возвращается минимальное значение дерева.

Этот дополнительный случай не оказывает влияния на асимптотическое время работы процедуры vEB-TREE-PREDECESSOR по сравнению с процедурой vEB-TREE-SUCCESSOR, так что в наихудшем случае время работы процедуры vEB-TREE-PREDECESSOR составляет $O(\lg \lg u)$.

Вставка элемента

Теперь рассмотрим вставку в дерево ван Эмде Боаса. Вспомним, что процедура PROTO-vEB-INSERT делает два рекурсивных вызова: один для вставки элемента, второй — для вставки в резюме номера кластера элемента. Процедура vEB-TREE-INSERT будет обходиться только одним рекурсивным вызовом. Как этого

добиться? Когда мы вставляем элемент, кластер, в который он попадает, либо уже имеет другой элемент, либо не имеет. Если в кластере уже есть другой элемент, то номер этого кластера уже находится в резюме, так что выполнять этот рекурсивный вызов нет необходимости. Если же кластер не содержит никакого другого элемента, то этот вставляемый элемент становится единственным элементом кластера, и нам не нужна рекурсия для того, чтобы вставить элемент в пустое vEB-дерево.

```
vEB-EMPTY-TREE-INSERT(V, x)
V.min = x
V.max = x
```

Имея эту вспомогательную процедуру, можно записать псевдокод процедуры vEB-TREE-INSERT(V, x), предполагающей, что x не является элементом, уже находящимся в множестве, которое представлено vEB-деревом V.

```
vEB-TREE-INSERT(V, x)
 1  if V.min == NIL
 2      vEB-EMPTY-TREE-INSERT(V, x)
 3  else if x < V.min
 4          Обменять x с V.min
 5      if V.u > 2
 6          if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
 7              vEB-TREE-INSERT(V.summary, high(x))
 8              vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
 9          else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
10      if x > V.max
11          V.max = x
```

Эта процедура работает следующим образом. В строке 1 выполняется проверка, не является ли V пустым vEB-деревом, и если это так, то этот простой случай обрабатывается строкой 2. В строках 3–11 предполагается, что V не пустое, а значит, некоторый элемент будет вставлен в один из кластеров V. Но этот элемент не обязательно представляет собой элемент x, переданный процедуре vEB-TREE-INSERT. Если $x < min$, что проверяется в строке 3, то x должен стать новым элементом min. Однако мы не хотим потерять исходный элемент min, так что нужно вставить его в один из кластеров V. В этом случае в строке 4 выполняется обмен x и min, так что мы вставляем в один из кластеров V исходный элемент min.

Строки 6–9 выполняются, только если V не является базовым vEB-деревом. В строке 6 выясняется, не является ли кластер для размещения элемента x в настоящее время пустым. Если это так, то строка 7 вставляет номер кластера x в резюме, а строка 8 обрабатывает простой случай вставки x в пустой кластер. Если кластер x в настоящее время не пустой, то строка 9 вставляет x в его кластер. В этом случае обновлять резюме не нужно, поскольку номер кластера x уже содержится в резюме.

Наконец строки 10 и 11 заботятся об обновлении max, если $x > max$. Заметим, что если V представляет собой непустое vEB-дерево в базовом случае, то строки 3 и 4, а также 10 и 11 корректно обновляют атрибуты min и max.

И вновь легко увидеть, как рекуррентное соотношение (20.4) описывает время работы рассматриваемой процедуры. В зависимости от результатов проверки в строке 6, выполняется либо рекурсивный вызов в строке 7 (для vEB-дерева с размером универсума $\sqrt[\downarrow]{u}$), либо рекурсивный вызов в строке 9 (для vEB-дерева с размером универсума $\sqrt[\uparrow]{u}$). В любом случае выполняется один рекурсивный вызов для vEB-дерева с размером универсума не более $\sqrt[\uparrow]{u}$. Поскольку остальная часть процедуры vEB-TREE-INSERT выполняется за время $O(1)$, применимо рекуррентное соотношение (20.4), и общее время работы составляет $O(\lg \lg u)$.

Удаление элемента

Наконец рассмотрим, как удалить элемент из дерева ван Эмде Боаса. В процедуре vEB-TREE-DELETE(V, x) предполагается, что x в настоящий момент является элементом множества, представленного vEB-деревом V.

```
vEB-TREE-DELETE(V, x)
if V.min == V.max
    V.min = NIL
    V.max = NIL
elseif V.u == 2
    if x == 0
        V.min = 1
    else V.min = 0
    V.max = V.min
else if x == V.min
        first-cluster = vEB-TREE-MINIMUM(V.summary)
        x = index(first-cluster,
            vEB-TREE-MINIMUM(V.cluster[first-cluster]))
        V.min = x
    vEB-TREE-DELETE(V.cluster[high(x)], low(x))
    if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
        vEB-TREE-DELETE(V.summary, high(x))
        if x == V.max
            summary-max = vEB-TREE-MAXIMUM(V.summary)
            if summary-max == NIL
                V.max = V.min
            else V.max = index(summary-max,
                    vEB-TREE-MAXIMUM(V.cluster[summary-max]))
    elseif x == V.max
        V.max = index(high(x),
            vEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

Процедура VEB-TREE-DELETE работает следующим образом. Если vEB-дерево V содержит только один элемент, то удалить его так же просто, как и вставить в пустое vEB-дерево: просто нужно установить min и max равными NIL. Этот случай обрабатывается в строках 1–3. В противном случае V содержит как минимум два элемента. В строке 4 проверяется, представляет ли собой V базовый случай, и если представляет, то в строках 5–8 атрибуты min и max устанавливаются равными одному остающемуся элементу.

В строках 9–22 предполагается, что V содержит два или более элементов и что $u \geq 4$. В этом случае необходимо удалить элемент из кластера. Однако элемент, который удаляется из кластера, может не быть элементом x, поскольку если x равен min, то, когда мы удалим x, новым min станет некоторый другой элемент из кластеров V, и потребуется удалить этот другой элемент из его кластера. Если проверка в строке 9 указывает на этот случай, строка 10 устанавливает *first-cluster* равным номеру кластера, который содержит наименьший элемент, отличный от min, а строка 11 устанавливает x равным значению наименьшего элемента в этом кластере. Этот элемент становится новым атрибутом min в строке 12, а поскольку мы присваиваем x его значение, то этот элемент и будет удален из своего кластера.

Когда мы достигаем строки 13, мы знаем, что должны удалить элемент x из его кластера независимо от того, является ли x исходным переданным процедуре VEB-TREE-DELETE значением или x представляет собой элемент, который становится новым минимумом. В строке 13 x удаляется из его кластера. Этот кластер может стать пустым, что и проверяется в строке 14, и если это так, то нужно удалить номер кластера x из резюме (что и происходит в строке 15). После обновления резюме может потребоваться обновить атрибут max. В строке 16 проверяется, не удаляем ли мы максимальный элемент V, и если это так, то в строке 17 *summary-max* получает номер непустого кластера с наибольшим номером. (Вызов VEB-TREE-MAXIMUM($V.summary$) работает, поскольку мы должны были рекурсивно вызвать VEB-TREE-DELETE для $V.summary$, и, таким образом, $V.summary.max$ уже должен быть обновлен (если в этом есть необходимость).) Если все кластеры V пустые, то единственный остающийся в V элемент — min; этот случай проверяется в строке 18, а в строке 19 выполняется соответствующее обновление max. В противном случае в строке 20 выполняется присваивание атрибуту max максимального элемента в непустом кластере с наибольшим номером. (Если элемент был удален из этого кластера, мы снова полагаемся на то, что рекурсивный вызов в строке 13 уже скорректировал атрибут max этого кластера.)

Наконец необходимо обработать случай, когда кластер x не стал пустым из-за удаления x. Хотя в этом случае обновлять резюме не нужно, может потребоваться обновление max. Этот случай проверяется в строке 21, и при необходимости обновить max это делается в строке 22 (и вновь мы полагаемся на рекурсивный вызов для коррекции max).

Теперь покажем, что в наихудшем случае время работы процедуры VEB-TREE-DELETE составляет $O(\lg\lg u)$. На первый взгляд, можно подумать, что рекуррентное соотношение (20.4) применимо не всегда, поскольку один вызов VEB-TREE-DELETE может сделать два рекурсивных вызова: один в строке 13, и вто-

рой — в строке 15. Хотя процедура может сделать оба рекурсивных вызова, давайте подумаем о том, что произойдет в таком случае. Чтобы был выполнен рекурсивный вызов в строке 15, проверка в строке 14 должна показать, что кластер x пуст. Единственный вариант, при котором кластер x может быть пустым — если x был единственным элементом этого кластера при выполнении рекурсивного вызова в строке 13. Но если x был единственным элементом своего кластера, то этот рекурсивный вызов выполняется за время $O(1)$, так как в нем выполняются только строки 1–3. Таким образом, есть две взаимоисключающие возможности.

- Рекурсивный вызов в строке 13 выполняется за константное время.
- Рекурсивный вызов в строке 15 не осуществляется.

В любом случае время выполнения процедуры vEB-TREE-DELETE характеризуется рекуррентным соотношением (20.4), а следовательно, в наихудшем случае оно составляет $O(\lg \lg u)$.

Упражнения

20.3.1

Модифицируйте vEB-дерево таким образом, чтобы оно поддерживало дублирующиеся ключи.

20.3.2

Модифицируйте vEB-дерево таким образом, чтобы оно поддерживало ключи со связанными сопутствующими данными.

20.3.3

Напишите псевдокод процедуры, создающей пустое дерево ван Эмде Боаса.

20.3.4

Что произойдет при вызове процедуры vEB-TREE-INSERT с элементом, уже имеющимся в vEB-дереве? Что произойдет при вызове процедуры vEB-TREE-DELETE с элементом, которого нет в vEB-дереве? Поясните поведение процедур в этих ситуациях. Покажите, как модифицировать vEB-деревья и операции над ними, чтобы за константное время можно было убеждаться в наличии в них определенных элементов.

20.3.5

Предположим, что вместо $\sqrt[\uparrow]{u}$ кластеров, каждый с размером универсума $\sqrt[\downarrow]{u}$, мы строим vEB-деревья с $u^{1/k}$ кластерами, каждый с размером универсума $u^{1-1/k}$, где $k > 1$ — константа. Каким будет время выполнения операций над таким деревом при их соответствующей модификации? Для простоты анализа считайте, что и $u^{1/k}$, и $u^{1-1/k}$ всегда являются целыми числами.

20.3.6

Создание vEB-дерева с размером универсума u требует времени $\Theta(u)$. Предположим, что необходимо явно подсчитать это время. Каково минимальное количество

операций n, для которого амортизированное время каждой операции над vEB-деревом составляет $O(\lg \lg u)$?

Задачи

20.1. Требования деревьев ван Эмде Боаса к памяти

В этой задаче исследуются требования деревьев ван Эмде Боаса к памяти и предлагается способ модификации структуры данных, при котором требуемая память зависит от количества n фактически хранящихся в дереве элементов, а не от размера универсума u. Для простоты считаем, что $\sqrt{u}$ всегда является целым числом.

а. Поясните, почему требуемая деревом ван Эмде Боаса с размером универсума u память $P(u)$ характеризуется следующим рекуррентным соотношением:

$$P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u}) . \qquad (20.5)$$

б. Докажите, что рекуррентное соотношение (20.5) имеет решение $P(u) = O(u)$.

Чтобы уменьшить требуемое количество памяти, определим ***дерево ван Эмде Боаса со сниженным количеством памяти*** (reduced-space van Emde Boas tree — ***RS-vEB-дерево***) как vEB-дерево V, но со следующими изменениями.

- Атрибут $V.cluster$ вместо простого массива указателей на vEB-деревья с размером универсума $\sqrt{u}$ представляет собой хеш-таблицу (см. главу 11), сохраненную как динамическая таблица (см. раздел 17.4). Хеш-таблица, соответствующая "массивной" версии $V.cluster$, хранит указатели на RS-vEB-деревья с размером универсума $\sqrt{u}$. Чтобы найти i-й кластер, мы ищем в хеш-таблице ключ i, так что кластер может быть найден с помощью единственной операции поиска в хеш-таблице.
- В хеш-таблице хранятся указатели только на непустые кластеры. Поиск в хеш-таблице пустого кластера возвращает значение NIL, указывающее, что искомый кластер пуст.
- Атрибут $V.summary$ в случае, когда все кластеры пустые, равен NIL. В противном случае $V.summary$ указывает на RS-vEB-дерево с размером универсума $\sqrt{u}$.

Поскольку хеш-таблица реализована с использованием динамической таблицы, требуемая для нее память пропорциональна количеству непустых кластеров.

Когда нужно вставить элемент в пустое RS-vEB-дерево, мы создаем последнее с помощью следующей процедуры, где параметр u представляет собой размер универсума RS-vEB-дерева.

```
CREATE-NEW-RS-VEB-TREE(u)
Выделить память для нового vEB-дерева V
V.u = u
V.min = NIL
V.max = NIL
V.summary = NIL
Создать V.cluster как пустую динамическую таблицу
return V
```

в. Модифицируйте процедуру VEB-TREE-INSERT таким образом, чтобы получить псевдокод процедуры RS-VEB-TREE-INSERT(V, x), которая вставляет x в RS-vEB-дерево V, вызывая при необходимости процедуру CREATE-NEW-RS-VEB-TREE.

г. Модифицируйте процедуру VEB-TREE-SUCCESSOR таким образом, чтобы получить псевдокод процедуры RS-VEB-TREE-SUCCESSOR(V, x), которая возвращает преемник элемента x в RS-vEB-дереве V или NIL, если у x нет преемника в V.

д. Докажите, что в предположении простого равномерного хеширования процедуры RS-VEB-TREE-INSERT и RS-VEB-TREE-SUCCESSOR имеют ожидаемое амортизированное время работы, равное $O(\lg\lg u)$.

е. В предположении, что элементы из vEB-дерева никогда не удаляются, докажите, что требуемая для RS-vEB-дерева память равна $O(n)$, где n — количество элементов, фактически хранящихся в RS-vEB-дереве.

ж. RS-vEB-деревья имеют еще одно преимущество перед vEB-деревьями: у них меньшее время создания. Сколько времени требуется, чтобы создать пустое RS-vEB-дерево?

20.2. y-быстрые лучи

В этой задаче исследуются "y-быстрые лучи" ("y-fast tries") Д. Уилларда (D. Willard), в которых, как и в деревьях ван Эмде Боаса, каждая из операций MEMBER, MINIMUM, MAXIMUM, PREDECESSOR и SUCCESSOR над элементами из универсума с размером u в наихудшем случае выполняются за время $O(\lg\lg u)$. Амортизированное время выполнения операций INSERT и DELETE также составляет $O(\lg\lg u)$. Подобно деревьям ван Эмде Боаса со сниженным количеством памяти (см. задачу 20.1), y-быстрые лучи используют для хранения n элементов только $O(n)$ памяти. В дизайне y-быстрых лучей используется идеальное хеширование (см. раздел 11.5).

Для разработки предварительной структуры предположим, что мы создаем идеальную хеш-таблицу, содержащую не только каждый элемент динамического множества, но и каждый префикс бинарного представления каждого элемента множества. Например, если $u = 16$, так что $\lg u = 4$, и $x = 13$ находится в множестве, то, поскольку бинарное представление 13 представляет собой 1101, идеаль-

ная хеш-таблица должна содержать строки 1, 11, 110 и 1101. В дополнение к хеш-таблице мы создаем дважды связанный список элементов, находящихся в данный момент в множестве, в порядке их возрастания.

а. Какое количество памяти требуется этой структуре данных?

б. Покажите, как выполнить операции MINIMUM и MAXIMUM за время $O(1)$; операции MEMBER, PREDECESSOR и SUCCESSOR — за время $O(\lg\lg u)$; операции INSERT и DELETE — за время $O(\lg u)$.

Для снижения требований к памяти до $O(n)$ внесем в структуру данных следующие изменения.

- Кластеризуем n элементов в $n/\lg u$ групп размером $\lg u$. (Будем считать, что n делится на $\lg u$.) Первая группа состоит из $\lg u$ наименьших элементов множества, вторая группа — из следующих по величине $\lg u$ элементов, и т.д.
- Объявим "представительное" значение для каждой группы. Представитель i-й группы как минимум такой же по величине, как наибольший элемент в i-й группе, и меньше любого элемента $(i+1)$-й группы. (Представителем последней группы может быть наибольший возможный элемент $u-1$.) Заметим, что представителем может быть и значение, в настоящий момент в множество не входящее.
- Будем хранить $\lg u$ элементов каждой группы в сбалансированном бинарном дереве поиска, таком как красно-черное дерево. Каждый представитель указывает на сбалансированное бинарное дерево поиска своей группы, а каждое сбалансированное бинарное дерево поиска указывает на представителя своей группы.
- В идеальной хеш-таблице хранятся только представители, которые также находятся в дважды связанном списке в возрастающем порядке.

Назовем такую структуру ***y-быстрым лучом***.

в. Покажите, что y-быстрый луч для хранения n элементов требует только $O(n)$ памяти.

г. Покажите, как выполнять операции MINIMUM и MAXIMUM в y-быстром луче за время $O(\lg\lg u)$.

д. Покажите, как выполнить операцию MEMBER за время $O(\lg\lg u)$.

е. Покажите, как выполнять операции PREDECESSOR и SUCCESSOR за время $O(\lg\lg u)$.

ж. Поясните, почему операции INSERT и DELETE требуют времени $\Omega(\lg\lg u)$.

з. Покажите, как ослабить требование, чтобы каждая группа в y-быстром луче имела ровно $\lg u$ элементов для того, чтобы амортизированное время работы операций INSERT и DELETE было равно $O(\lg\lg u)$, и при этом не повлиять на асимптотические времена работы других операций.

Заключительные замечания

Структура данных в этой главе названа по имени П. ван Эмде Боаса (P. van Emde Boas), который изложил соответствующую идею в 1975 году [337]. В более поздних статьях ван Эмде Боаса [338] и ван Эмде Боаса, Кааса (Kaas) и Зейлстры (Zijlstra) [339] эта идея была развита и уточнена. Впоследствии Мельхорн (Mehlhorn) и Няхер (Näher) [250] расширили эту идею для простых размеров универсумов. В книге Мельхорна [247] используется несколько иной подход к деревьям ван Эмде Боаса, чем подход из данной главы.

Используя идеи, лежащие в основе деревьев ван Эмде Боаса, Дементьев (Dementiev) и др. [82] разработали нерекурсивное трехуровневое дерево поиска, которое в их экспериментах по производительности превосходило деревья ван Эмде Боаса.

Ванг (Wang) и Лин (Lin) [345] разработали версии деревьев с использованием аппаратной конвейеризации, с константным амортизированным временем работы операций и $O(\lg \lg u)$ стадиями конвейера.

Патраску (Pătraşcu) и Торуп (Thorup) [271, 272] показали, что нижняя граница времени поиска предшественника в дереве ван Эмде Боаса оптимальна (даже в случае разрешенной рандомизации).

Глава 21. Структуры данных для непересекающихся множеств

В некоторых приложениях выполняется группировка n различных элементов в набор непересекающихся множеств. Две важные операции, которые должны выполняться с таким набором, — поиск множества, содержащего заданный элемент, и объединение двух множеств. В данной главе мы познакомимся со структурой данных, которая поддерживает эти операции.

В разделе 21.1 описаны операции, поддерживаемые указанной структурой данных, и представлено простое приложение. В разделе 21.2 мы рассмотрим использование простого связанного списка для представления связанных множеств. Более эффективное представление с помощью корневых деревьев приведено в разделе 21.3. Время работы с использованием деревьев линейно для всех практических применений, хотя теоретически оно сверхлинейно. В разделе 21.4 определяется и обсуждается очень быстро растущая функция и ее очень медленно растущая инверсия, которая и проявляется во времени работы операций над представлением непересекающихся множеств с использованием деревьев. Там же с помощью амортизационного анализа доказывается верхняя сверхлинейная граница времени работы.

21.1. Операции над непересекающимися множествами

Структура данных для непересекающихся множеств (disjoint-set data structure) поддерживает набор $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ непересекающихся множеств. Каждое множество идентифицируется ***представителем*** (representative), который представляет собой некоторый член множества. В одних приложениях не имеет значения, какой именно элемент множества используется в качестве представителя; главное, чтобы при запросе представителя множества дважды, без внесения изменений в множество между запросами, возвращался один и тот же элемент. В других приложениях может действовать предопределенное правило выбора представителя, например наименьшего члена множества (само собой разумеется, в предположении о возможности упорядочения элементов множества).

Как и в других изученных нами реализациях динамических множеств, каждый элемент множества представляет некоторый объект x. Мы хотим обеспечить поддержку следующих операций.

MAKE-SET(x) создает новое множество, состоящее из одного члена (который, соответственно, является его представителем) x. Поскольку множества непересекающиеся, требуется, чтобы x не входил ни в какое иное множество.

UNION(x, y) объединяет динамические множества, которые содержат x и y (обозначим их через S_x и S_y), в новое множество. Предполагается, что до выполнения операции указанные множества не пересекались. Представителем образованного в результате множества является произвольный элемент $S_x \cup S_y$, хотя многие реализации операции UNION выбирают новым представителем представителя множества S_x или S_y. Поскольку нам необходимо, чтобы все множества были непересекающимися, операция UNION концептуально должна уничтожать множества S_x и S_y, удаляя их из коллекции $\mathcal{S}$.

FIND-SET(x) возвращает указатель на представителя (единственного) множества, в котором содержится элемент x.

В этой главе мы проанализируем зависимость времени работы структуры данных для непересекающихся множеств от двух параметров: количества операций MAKE-SET n и общего количества операций MAKE-SET, UNION и FIND-SET m. Поскольку множества непересекающиеся, каждая операция UNION уменьшает количество множеств на 1. Следовательно, после $n - 1$ операций UNION останется только одно множество, так что количество операций UNION не может превышать $n - 1$. Заметим также, что поскольку в общее количество операций m включены операции MAKE-SET, то $m \geq n$. Предполагается также, что n операций MAKE-SET являются первыми n выполненными операциями.

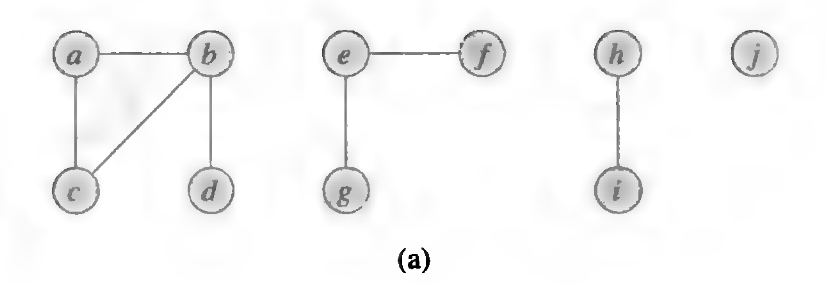
Приложение структур данных для непересекающихся множеств

Одно из многих применений структур данных для непересекающихся множеств — в задаче об определении связных компонентов неориентированного графа (см. раздел Б.4). Так, на рис. 21.1,(а) показан граф, состоящий из четырех связных компонентов.

Процедура CONNECTED-COMPONENTS, приведенная ниже, использует операции над непересекающимися множествами для вычисления связных компонентов графа. После того как процедура CONNECTED-COMPONENTS разобьет множество вершин графа на непересекающиеся множества, процедура SAME-COMPONENT в состоянии определить, принадлежат ли две данные вершины одному и тому же связному компоненту[1]. (Множество вершин графа G обозначим как $G.V$, а множество ребер — как $G.E$.)

---

[1]Если ребра графа являются "статическими", т.е. не изменяются с течением времени, то вычислить связные компоненты можно быстрее с помощью поиска в глубину (см. упр. 22.3.12). Однако иногда ребра добавляются "динамически", и нам нужно поддерживать связные компоненты при добавлении нового ребра. В этой ситуации приведенная здесь реализация оказывается эффективнее, чем повторное выполнение поиска вглубь для каждого нового ребра.



(а)

Обработанные ребра	Набор непересекающихся множеств									
Исходные множества	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(б)

Рис. 21.1. **(а)** Граф с четырьмя связными компонентами: $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i\}$ и $\{j\}$. **(б)** Набор непересекающихся множеств после обработки каждого ребра.

```
CONNECTED-COMPONENTS(G)
for каждой вершины v ∈ G.V
    MAKE-SET(v)
for каждого ребра (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

```
SAME-COMPONENT(u, v)
if FIND-SET(u) == FIND-SET(v)
    return TRUE
else return FALSE
```

Процедура CONNECTED-COMPONENTS сначала помещает каждую вершину v в ее собственное множество. Затем для каждого ребра (u, v) выполняется объединение множеств, содержащих u и v. В соответствии с упр. 21.1.2 после обработки всех ребер две вершины будут находиться в одном связном компоненте тогда и только тогда, когда соответствующие объекты находятся в одном множестве. Таким образом, процедура CONNECTED-COMPONENTS вычисляет множества так, что процедура SAME-COMPONENT может определить, находятся ли две вершины в одиом и том же связном компоненте. На рис. 21.1, (б) показан процесс вычисления непересекающихся множеств процедурой CONNECTED-COMPONENTS.

В реальной реализации описанного алгоритма представления графа и структуры непересекающихся множеств требуют наличия взаимных ссылок, т.е. объ-

ект, представляющий вершину, должен содержать указатель на соответствующий объект в непересекающемся множестве и наоборот. Эти детали программной реализации зависят от используемого для реализации языка программирования и здесь не рассматриваются.

Упражнения

21.1.1

Предположим, что процедура CONNECTED-COMPONENTS вызывается для неориентированного графа $G = (V, E)$, где $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, а ребра из E обрабатываются в следующем порядке: $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e)$. Перечислите вершины в каждом связном компоненте после выполнения каждой итерации в строках 3–5.

21.1.2

Покажите, что после того, как все ребра будут обработаны процедурой CONNECTED-COMPONENTS, две вершины находятся в одном связном компоненте тогда и только тогда, когда они находятся в одном и том же множестве.

21.1.3

Сколько раз вызывается процедура FIND-SET в процессе выполнения процедуры CONNECTED-COMPONENTS над неориентированным графом $G = (V, E)$ с k связными компонентами? Сколько раз вызывается процедура UNION? Выразите ваш ответ через $|V|$, $|E|$ и k.

21.2. Представление непересекающихся множеств с помощью связанных списков

На рис. 21.2, (а) показан простейший способ реализации структуры данных для непересекающихся множеств: каждое множество представлено своим связанным списком. Объект каждого множества имеет атрибуты *head*, указывающий на первый объект списка, и *tail*, указывающий на последний объект списка. Каждый объект списка содержит член множества, указатель на следующий объект в списке и указатель на объект множества. Объекты в пределах каждого списка могут располагаться в любом порядке. Представителем множества является член в первом объекте списка.

При использовании такого представления в виде связанных списков процедуры MAKE-SET и FIND-SET легко реализуются и время их работы равно $O(1)$. Процедура MAKE-SET(x) создает новый связанный список с единственным объектом x, а процедура FIND-SET(x) просто следует по указателю на объект множества и возвращает член объекта, на который указывает *head*. Например, на рис. 21.2, (а) вызов FIND-SET(g) вернет f.

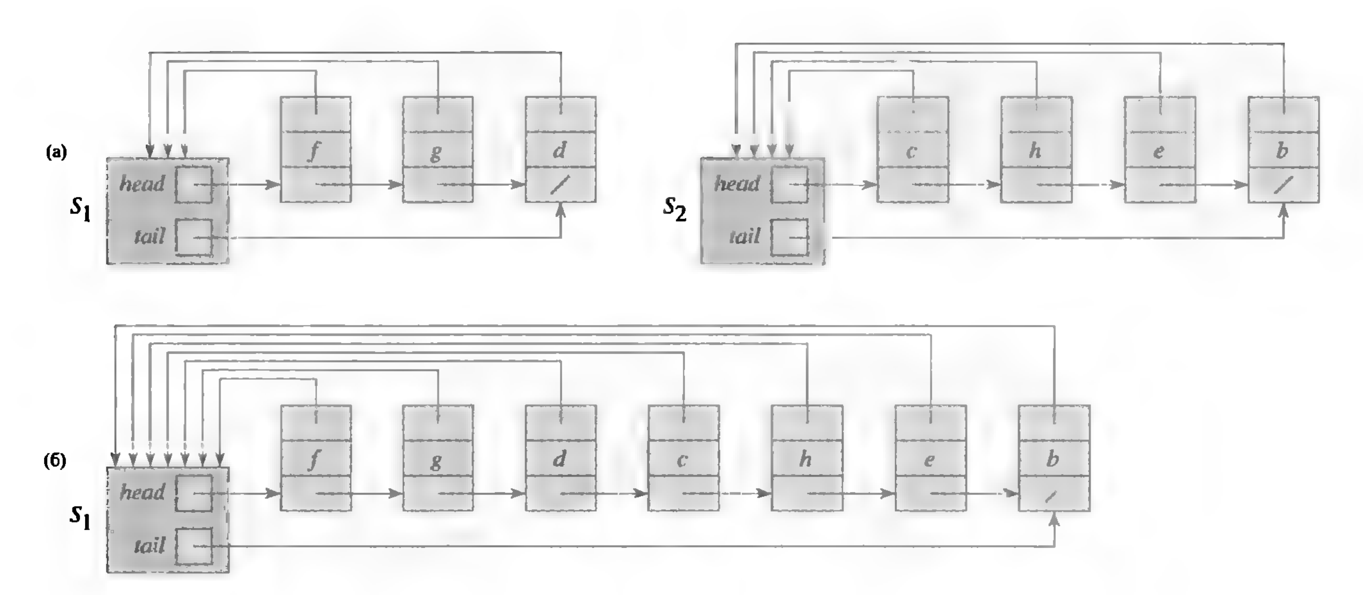


Рис. 21.2. **(а)** Представление двух множеств в виде связанных списков. Множество S_1 содержит члены d, f и g, с представителем f, а множество S_2 содержит члены b, c, e и h, с представителем c. Каждый объект в списке содержит член множества, указатель на следующий объект в списке и указатель на объект множества. Каждый объект множества имеет указатели *head* и *tail* на первый и последний объекты соответственно. **(б)** Результат операции UNION(g, e), которая добавляет связанный список, содержащий e, к связанному списку, содержащему g. Представителем полученного в результате множества является f. Объект множества S_2 для списка e уничтожается.

Простая реализация объединения

Простейшая реализация операции UNION при использовании связанных списков требует гораздо больше времени, чем процедуры MAKE-SET и FIND-SET. Как показано на рис. 21.2, (б), мы выполняем процедуру UNION(x, y), добавляя список с элементом x в конец списка, содержащего y. Представитель списка x становится представителем результирующего списка. Мы используем указатель *tail* для списка с элементом x для того, чтобы быстро определить, куда следует добавить список, содержащий y. Поскольку все члены списка y объединяются со списком x, можно уничтожить объект множества для списка y. К сожалению, требуется обновить указатель на объект множества для каждого объекта, исходно входящего в список y, что требует времени, линейно зависящего от длины списка y. На рис. 21.2, например, операция UNION(g, e) требует обновления указателей объектов b, c, e и h.

Действительно, нетрудно построить последовательность из m операций над n объектами, которая требует $\Theta(n^2)$ времени. Предположим, что у нас есть объекты $x_1, x_2, \ldots, x_n$. Мы выполняем последовательность из n операций MAKE-SET, за которой следует последовательность из $n - 1$ операций UNION, показанных на рис. 21.3, так что $m = 2n - 1$. На выполнение n операций MAKE-SET мы тратим время $\Theta(n)$. Поскольку i-я операция UNION обновляет i объектов, общее количество объектов, обновленных всеми $n - 1$ операциями UNION, равно

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

Операция	Количество обновленных объектов
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$

Рис. 21.3. Последовательность $2n-1$ операций над n объектами, которая требует $\Theta(n^2)$ времени (в среднем $\Theta(n)$ на одну операцию) при представлении с использованием связанных списков и простой реализации процедуры UNION.

Общее количество операций равно $2n-1$, так что каждая операция в среднем требует для выполнения времени $\Theta(n)$. Таким образом, амортизированное время выполнения операции составляет $\Theta(n)$.

Весовая эвристика

В наихудшем случае представленная реализация процедуры UNION требует в среднем $\Theta(n)$ времени на один вызов, поскольку может оказаться, что мы присоединяем длинный список к короткому и должны при этом обновить поля указателей на представителя всех членов длинного списка. Предположим теперь, что каждый список включает также поле длины списка (которое легко поддерживается) и что мы всегда добавляем меньший список к большему (при одинаковых длинах порядок добавления не имеет значения). При такой простейшей ***весовой эвристике*** (weighted-union heuristic) одна операция UNION может потребовать время $\Omega(n)$, если оба множества имеют по $\Omega(n)$ членов. Однако, как показывает следующая теорема, последовательность из m операций MAKE-SET, UNION и FIND-SET, n из которых составляют операции MAKE-SET, требует для выполнения время $O(m + n \lg n)$.

Теорема 21.1

При использовании связанных списков для представления непересекающихся множеств и применении весовой эвристики последовательность из m операций MAKE-SET, UNION и FIND-SET, n из которых составляют операции MAKE-SET, требует для выполнения время $O(m + n \lg n)$.

Доказательство. Поскольку каждая операция UNION объединяет два непересекающихся множества, всего выполняется не более $n-1$ операции UNION. Теперь вычислим границу для общего времени, требуемого для выполнения этих операций UNION. Начнем с вычисления верхней границы количества обновлений указателей на объект множества для каждого из n элементов. Рассмотрим конкретный объект x. Мы знаем, что всякий раз, когда происходит обновление

указателя в объекте x, он должен находиться в меньшем из объединяемых множеств. Следовательно, когда происходит первое обновление указателя в объекте x, образованное в результате множество содержит не менее двух элементов. При следующем обновлении указателя на представителя в объекте x полученное после объединения множество содержит не менее четырех членов. Продолжая рассуждения, приходим к выводу о том, что для произвольного $k \leq n$, после того как указатель в объекте x был обновлен $\lceil \lg k \rceil$ раз, полученное в результате множество должно иметь не менее k элементов. Поскольку максимальное множество может иметь только n элементов, во всех операциях UNION указатель каждого объекта может быть обновлен не более $\lceil \lg n \rceil$ раз. Мы должны также учесть обновления указателей *tail* и длин списков, для выполнения которых при каждой операции UNION требуется $\Theta(1)$ времени. Таким образом, общее время, необходимое для обновления n объектов, составляет $O(n \lg n)$.

Отсюда легко выводится время, необходимое для всей последовательности из m операций. Каждая операция MAKE-SET и FIND-SET занимает время $O(1)$, а всего их — $O(m)$. Таким образом, полное время выполнения всей последовательности операций — $O(m + n \lg n)$. ■

Упражнения

21.2.1

Напишите псевдокод процедур MAKE-SET, FIND-SET и UNION с использованием связанных списков и весовой эвристики. Не забудьте указать атрибуты, которые должны быть у объектов множеств и объектов списков.

21.2.2

Покажите, какая образуется структура данных и какие ответы дают операции FIND-SET в следующей программе. Используются представление с помощью связанных списков и весовая эвристика.

```
for i = 1 to 16
    MAKE-SET(x_i)
for i = 1 to 15 by 2
    UNION(x_i, x_{i+1})
for i = 1 to 13 by 4
    UNION(x_i, x_{i+2})
UNION(x_1, x_5)
UNION(x_11, x_13)
UNION(x_1, x_10)
FIND-SET(x_2)
FIND-SET(x_9)
```

Считаем, что если множества, содержащие x_i и x_j, имеют один и тот же размер, то операция UNION(x_i, x_j) присоединяет список x_j к списку x_i.

21.2.3

Адаптируйте доказательство теоремы 21.1 для получения границ амортизированного времени выполнения процедур MAKE-SET и FIND-SET, равного $O(1)$, и амортизированного времени выполнения процедуры UNION, равного $O(\lg n)$, при использовании связанных списков и весовой эвристики.

21.2.4

Найдите точную асимптотическую границу времени работы последовательности операций на рис. 21.3 в предположении использования связанных списков и весовой эвристики.

21.2.5

Профессор подозревает, что в каждом объекте множества можно поддерживать только один указатель вместо двух (*head* и *tail*), в то время как в каждом элементе списка их количество равно двум. Покажите, что подозрения профессора небезосновательны, описав, как представить каждое множество связанным списком, таким, что каждая операция имеет то же время работы, что и указанное в тексте раздела. Опишите также, как работают эти операции. Ваша схема должна использовать эвристику с тем же эффектом, что и описанный в тексте раздела. (*Указание:* используйте в качестве представителя множества хвост списка.)

21.2.6

Предложите простое изменение процедуры UNION для связанных списков, которое избавляет от необходимости поддерживать указатель *tail* на последний объект каждого списка. Ваше предложение не должно изменить асимптотическое время работы процедуры UNION независимо от того, используется ли в ней весовая эвристика. (*Указание:* вместо присоединения одного списка к другому воспользуйтесь вставкой в начало списка.)

21.3. Леса непересекающихся множеств

В более быстрой реализации непересекающихся множеств мы представляем множества в виде корневых деревьев, каждый узел которых содержит один член множества, а каждое дерево представляет одно множество. В ***лесу непересекающихся множеств*** (disjoint-set forest) (рис. 21.4, (а)) каждый член указывает только на родительский узел. Корень каждого дерева содержит представителя и является родительским узлом для самого себя. Как мы увидим далее, хотя наивное программирование и не приводит к более эффективной работе по сравнению со связанными списками, введение двух эвристик — "объединение по рангу" и "сжатие пути" — позволяет достичь асимптотически оптимального выполнения операций над непересекающимися множествами.

Три операции над непересекающимися множествами выполняются следующим образом. Операция MAKE-SET просто создает дерево с одним узлом. Поиск в операции FIND-SET выполняется простым перемещением до корня дерева по

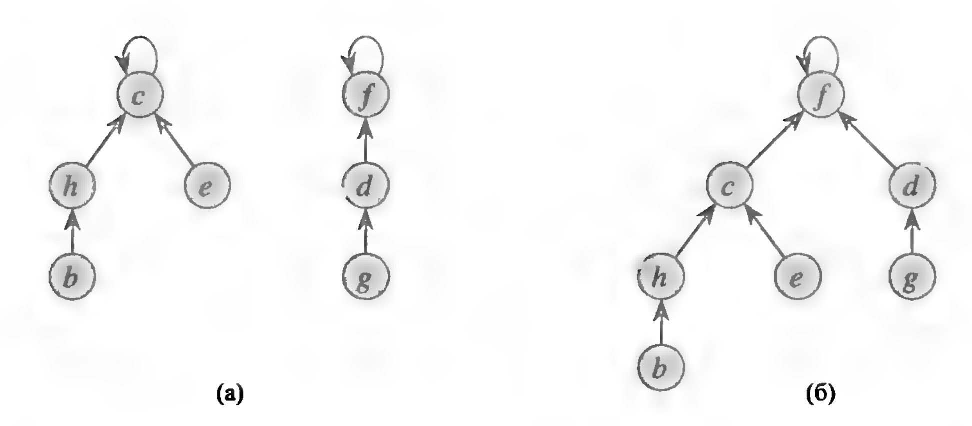


Рис. 21.4. Лес непересекающихся множеств. **(а)** Два дерева представляют два множества, показанных на рис. 21.2. Дерево слева представляет множество $\{b, c, e, h\}$, где c является представителем, а дерево справа представляет множество $\{d, f, g\}$ с представителем f. **(б)** Результат выполнения UNION(e, g).

указателям на родительские узлы. Посещенные узлы на этом простом пути составляют ***путь поиска*** (find path). Операция UNION, показанная на рис. 21.4, (б), состоит в том, что корень одного дерева указывает на корень другого.

Эвристики для снижения времени работы

Пока что у нас нет никаких особых преимуществ перед реализацией с использованием связанных списков: последовательность из $n - 1$ операций UNION может создать дерево, которое представляет собой линейную цепочку из n узлов. Однако мы можем воспользоваться двумя эвристиками и достичь практически линейного времени работы от общего количества операций m.

Первая эвристика, ***объединение по рангу*** (union by rank), аналогична весовой эвристике при использовании связанных списков. Ее суть заключается в том, чтобы корень дерева с меньшим количеством узлов указывал на корень дерева с бóльшим количеством узлов. Вместо явной поддержки размера поддерева для каждого узла можно воспользоваться подходом, который упростит анализ, — использовать ***ранг*** (rank) каждого корня, который представляет собой верхнюю границу высоты узла. При выполнении операции UNION корень с меньшим рангом должен указывать на корень с бóльшим рангом.

Вторая эвристика, ***сжатие пути*** (path compression), также достаточно проста и эффективна. Как показано на рис. 21.5, она используется в процессе выполнения операции FIND-SET и делает каждый узел указывающим непосредственно на корень. Сжатие пути не изменяет ранги узлов.

Псевдокод для работы с лесами непересекающихся множеств

Для реализации леса непересекающихся множеств с применением эвристики объединения по рангу мы должны отслеживать ранг узлов. Для каждого узла x нами поддерживается целочисленное значение $x.rank$, которое представляет собой верхнюю границу высоты x (количество ребер на самом длинном простом пути от x к листу, являющемуся его потомком). При создании множества из одного элемента процедурой MAKE-SET начальный ранг узла в соответствующем

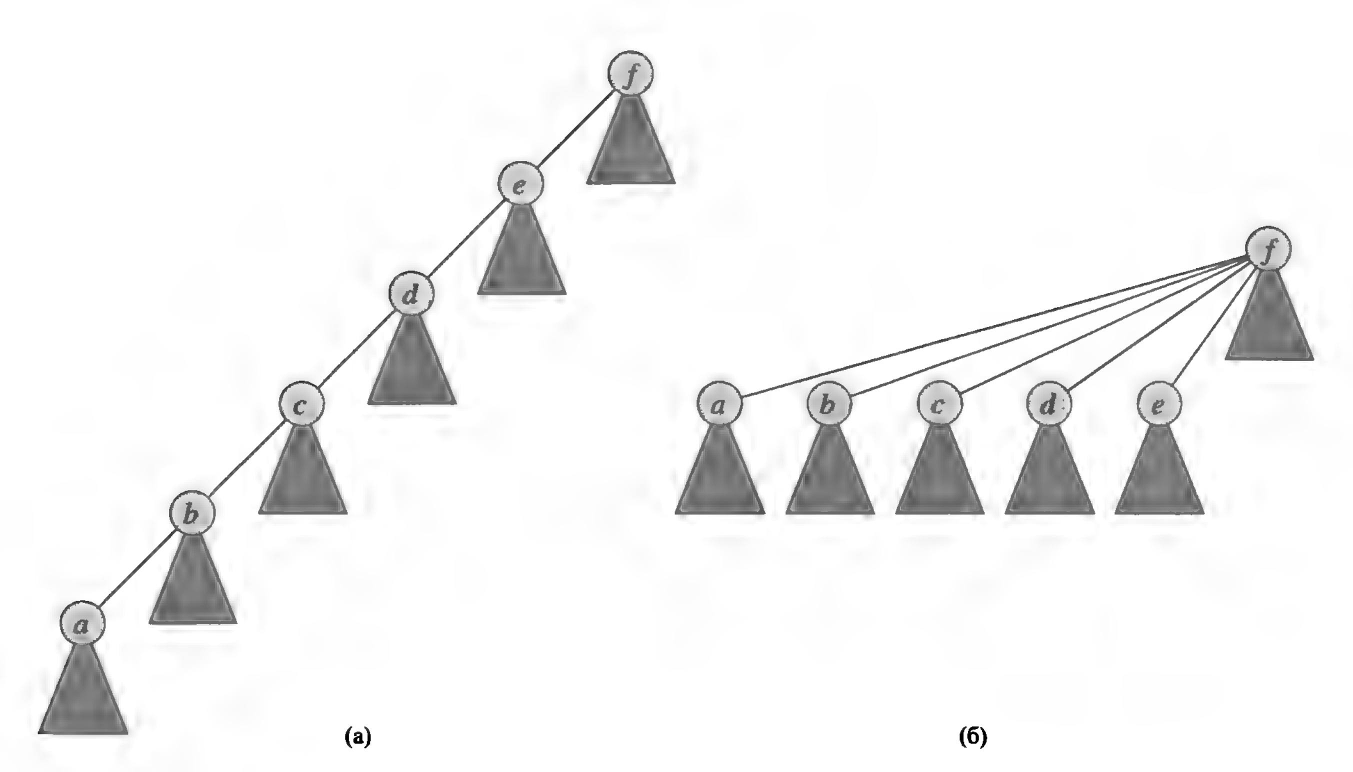


Рис. 21.5. Сжатие пути в процессе операции FIND-SET. Стрелки и петли в корнях опущены. **(а)** Дерево, представляющее множество, до выполнения операции FIND-SET(a). Треугольники представляют поддеревья, корнями которых являются показанные узлы. Каждый узел имеет указатель на своего родителя. **(б)** То же множество после выполнения операции FIND-SET(a). Каждый узел на пути поиска теперь указывает непосредственно на корень.

дереве устанавливается равным 0. Операции FIND-SET оставляют ранги неизменными. При применении процедуры UNION для объединения двух деревьев имеются две ситуации, зависящие от того, имеют объединяемые деревья одинаковый ранг или нет. Если ранги деревьев разные, мы делаем корень с бóльшим рангом родительским узлом по отношению к корню с меньшим рангом, но сами ранги при этом остаются неизменными. Если же корни имеют одинаковые ранги, то мы произвольным образом выбираем один из корней в качестве родительского и увеличиваем его ранг.

Переведем этот метод в псевдокод. Родительский по отношению к x узел обозначается как $x.p$. Процедура LINK, являющаяся подпрограммой, вызываемой процедурой UNION, получает в качестве входных данных указатели на два корня.

```
MAKE-SET(x)
x.p = x
x.rank = 0
```

```
UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank == y.rank
5          y.rank = y.rank + 1
```

Процедура FIND-SET со сжатием пути достаточно проста.

```
FIND-SET(x)
1  if x ≠ x.p
2      x.p = FIND-SET(x.p)
3  return x.p
```

Процедура FIND-SET является ***двухпроходной***: при первом проходе она ищет путь к корню дерева, а при втором проходе в обратном направлении по найденному пути происходит обновление узлов, которые теперь указывают непосредственно на корень дерева. Каждый вызов FIND-SET(x) возвращает $x.p$ в строке 3. Если x — корень, то строка 2 не выполняется и возвращается значение $x.p$, равное x, и на этом рекурсия завершается. В противном случае выполнение строки 2 приводит к рекурсивному вызову с параметром $x.p$, который возвращает указатель на корень. В той же строке 2 происходит обновление узла x, после которого он указывает непосредственно на найденный корень, и этот указатель возвращается вызывающей процедуре в строке 3.

Влияние эвристик на время работы

Будучи примененными раздельно, объединение по рангу и сжатие пути приводят к повышению эффективности операций над лесом непересекающихся множеств. Еще больший выигрыш дает совместное применение этих эвристик. Объединение по рангу само по себе дает время работы $O(m \lg n)$ (см. упр. 21.4.4), причем эта оценка не может быть улучшена (см. упр. 21.3.3). Хотя мы не будем доказывать это здесь, если имеется n операций MAKE-SET (а следовательно, не более $n-1$ операций UNION) и f операций FIND-SET, то сжатие пути само по себе приводит ко времени работы в наихудшем случае $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

При совместном использовании обеих эвристик время работы в наихудшем случае составляет $O(m\,\alpha(n))$, где $\alpha(n)$ — *очень* медленно растущая функция, которую мы определим в разделе 21.4. Для любого мыслимого приложения с использованием непересекающихся множеств $\alpha(n) \le 4$, так что можно рассматривать время работы во всех практических ситуациях как линейно зависящее от m (хотя, строго говоря, оно сверхлинейное). В разделе 21.4 мы докажем эту верхнюю границу.

Упражнения

21.3.1

Выполните упр. 21.2.2 для леса непересекающихся множеств с использованием объединения по рангу и сжатия пути.

21.3.2

Напишите нерекурсивную версию процедуры FIND-SET со сжатием пути.

21.3.3

Приведите последовательность из m операций MAKE-SET, UNION и FIND- SET, n из которых — операции MAKE-SET, время выполнения которой составляет $\Omega(m \lg n)$ при использовании только объединения по рангу.

21.3.4

Предположим, что мы хотим добавить операцию PRINT-SET(x), которая для заданного узла x выводит все члены множества x в произвольном порядке. Покажите, как можно добавить к каждому узлу леса непересекающихся множеств по одному атрибуту, так чтобы время работы процедуры PRINT-SET(x) линейно зависело от количества членов множества x и при этом асимптотические времена работы других операций остались неизменными. Считайте, что каждый член множества выводится за время $O(1)$.

21.3.5 ★

Покажите, что время выполнения произвольной последовательности m операций MAKE-SET, FIND-SET и LINK, в которой все операции LINK выполняются до первой операции FIND-SET, равно $O(m)$, если используются как объединение по рангу, так и сжатие пути. Что будет, если воспользоваться только сжатием пути?

★ 21.4. Анализ объединения по рангу со сжатием пути

Как отмечалось в разделе 21.3, время работы m операций над непересекающимися множествами из n элементов при использовании объединения по рангу и сжатия пути равно $O(m\,\alpha(n))$. В данном разделе мы рассмотрим функцию α и выясним, насколько медленно она растет. Затем мы докажем приведенное выше время работы с использованием метода потенциала из амортизационного анализа.

Очень быстро растущая функция и очень медленно растущая функция, обратная к ней

Определим для целых $k \geq 0$ и $j \geq 1$ функцию $A_k(j)$ как

$$A_k(j) = \begin{cases} j+1\,, & \text{если } k = 0\,, \\ A_{k-1}^{(j+1)}(j)\,, & \text{если } k \geq 1\,, \end{cases}$$

где выражение $A_{k-1}^{(j+1)}(j)$ использует функционально-итеративные обозначения из раздела 3.2. В частности, $A_{k-1}^{(0)}(j) = j$, а $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ для $i \geq 1$. Параметр k будем называть ***уровнем*** функции A.

Функция $A_k(j)$ — строго возрастающая по j и k. Для того чтобы увидеть, насколько быстро растет данная функция, начнем с записи функций $A_1(j)$ и $A_2(j)$ в явном виде.

Лемма 21.2

Для любого целого $j \geq 1$ мы имеем $A_1(j) = 2j + 1$.

Доказательство. Воспользуемся индукцией по i, чтобы показать, что $A_0^{(i)}(j) = j+i$. Для базы индукции мы имеем $A_0^{(0)}(j) = j = j+0$. Для выполнения шага индукции предположим, что $A_0^{(i-1)}(j) = j+(i-1)$. Тогда $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j+(i-1))+1 = j+i$. Наконец заметим, что $A_1(j) = A_0^{(j+1)}(j) = j+(j+1) = 2j+1$. ■

Лемма 21.3

Для любого целого $j \geq 1$ мы имеем $A_2(j) = 2^{j+1}(j+1) - 1$.

Доказательство. Воспользуемся индукцией по i, чтобы показать, что $A_1^{(i)}(j) = 2^i(j+1) - 1$. Для базы индукции мы имеем $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. Для выполнения шага индукции предположим, что $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Тогда $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1)-1) = 2\cdot(2^{i-1}(j+1)-1)+1 = 2^i(j+1)-2+1 = 2^i(j+1)-1$. Наконец заметим, что $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1)-1$. ■

Теперь посмотрим, насколько быстро растет функция $A_k(j)$, просто вычисляя $A_k(1)$ для уровней $k = 0,1,2,3,4$. Из определения $A_0(k)$ и рассмотренных выше лемм мы имеем $A_0(1) = 1+1 = 2$, $A_1(1) = 2\cdot 1+1 = 3$ и $A_2(1) = 2^{1+1}\cdot(1+1) - 1 = 7$. Кроме того,

$$\begin{aligned}
A_3(1) &= A_2^{(2)}(1) \\
&= A_2(A_2(1)) \\
&= A_2(7) \\
&= 2^8 \cdot 8 - 1 \\
&= 2^{11} - 1 \\
&= 2047
\end{aligned}$$

и

$$\begin{aligned}
A_4(1) &= A_3^{(2)}(1) \\
&= A_3(A_3(1)) \\
&= A_3(2047) \\
&= A_2^{(2048)}(2047) \\
&\gg A_2(2047) \\
&= 2^{2048} \cdot 2048 - 1 \\
&> 2^{2048} \\
&= (2^4)^{512} \\
&= 16^{512} \\
&\gg 10^{80} ,
\end{aligned}$$

что представляет собой оценочное количество атомов в наблюдаемой Вселенной. (Символ "$\gg$" означает отношение "гораздо больше, чем".)

Определим обратную к $A_k(n)$ функцию для $n \geq 0$ следующим образом:

$$\alpha(n) = \min \{k : A_k(1) \geq n\} .$$

Другими словами, $\alpha(n)$ — наименьший уровень k, для которого $A_k(1)$ не меньше n. Исходя из найденных выше значений $A_k(1)$ мы видим, что

$$\alpha(n) = \begin{cases} 0 & \text{для } 0 \leq n \leq 2 , \\ 1 & \text{для } n = 3 , \\ 2 & \text{для } 4 \leq n \leq 7 , \\ 3 & \text{для } 8 \leq n \leq 2047 , \\ 4 & \text{для } 2048 \leq n \leq A_4(1) . \end{cases}$$

Только для невероятно больших, "астрономических" значений n (бóльших $A_4(1)$) значение функции $\alpha(n)$ превышает 4, так что для всех практических применений $\alpha(n) \leq 4$.

Свойства рангов

В оставшейся части этого раздела мы докажем, что при использовании объединения по рангам и сжатия пути граница времени работы операций над непересекающимися множествами равна $O(m\,\alpha(n))$. Для этого нам потребуются некоторые простые свойства рангов.

Лемма 21.4

Для всех узлов x мы имеем $x.rank \leq x.p.rank$, причем при $x \neq x.p$ выполняется строгое неравенство. Значение $x.rank$ изначально равно 0 и со временем

увеличивается, пока не будет достигнуто $x \neq x.p$; после этого величина $x.rank$ не изменяется. Значение $x.p.rank$ монотонно растет со временем.

Доказательство. Доказательство выполняется простой индукцией по количеству операций с использованием реализаций процедур MAKE-SET, UNION и FIND-SET в разделе 21.3. Подробности доказательства оставлены в качестве упр. 21.4.1. ■

Следствие 21.5

При перемещении вдоль простого пути от произвольного узла к корню ранг строго возрастает. ■

Лемма 21.6

Ранг любого узла не превышает $n - 1$.

Доказательство. Начальный ранг каждого узла — 0, и он возрастает только при выполнении операции LINK. Поскольку выполняется не более $n - 1$ операции UNION, операций LINK также выполняется не более чем $n - 1$. Поскольку каждая операция LINK либо оставляет все ранги неизменными, либо увеличивает ранг некоторого узла на 1, ни один ранг не может превышать $n - 1$. ■

Лемма 21.6 дает слабую оценку границы рангов. В действительности ранг любого узла не превосходит $\lfloor \lg n \rfloor$ (см. упр. 21.4.2). Однако для наших целей достаточно границы, определяемой леммой 21.6.

Доказательство границы времени работы

Для доказательства границы $O(m\alpha(n))$ мы воспользуемся методом потенциала из амортизационного анализа (см. раздел 17.3). В ходе амортизационного анализа удобно считать, что мы выполняем операцию LINK, а не UNION. Иначе говоря, поскольку параметрами процедуры LINK являются указатели на два корня, мы считаем, что соответствующие операции FIND-SET выполняются отдельно. Приведенная далее лемма показывает, что, даже если учесть дополнительные операции FIND-SET, инициированные вызовами UNION, асимптотическое время работы останется неизменным.

Лемма 21.7

Предположим, что мы преобразуем последовательность S' из m' операций MAKE-SET, UNION и FIND-SET в последовательность S из m операций MAKE-SET, LINK и FIND-SET путем преобразования каждой операции UNION в две операции FIND-SET, за которыми следует операция LINK. Тогда, если последовательность S выполняется за время $O(m\,\alpha(n))$, то последовательность S' выполняется за время $O(m'\,\alpha(n))$.

Доказательство. Поскольку каждая операция UNION в последовательности S' преобразуется в три операции в S, выполняется соотношение $m' \leq m \leq 3m'$.

Так как $m = O(m')$, из границы времени работы $O(m\,\alpha(n))$ преобразованной последовательности S следует граница $O(m'\,\alpha(n))$ времени работы исходной последовательности S'. ■

В оставшейся части этого раздела мы полагаем, что исходная последовательность из m' операций MAKE-SET, UNION и FIND-SET преобразована в последовательность из m операций MAKE-SET, LINK и FIND-SET. Теперь докажем, что время выполнения полученной последовательности равно $O(m\,\alpha(n))$, и обратимся к лемме 21.7 для доказательства времени работы $O(m'\,\alpha(n))$ исходной последовательности из m' операций.

Функция потенциала

Используемая нами функция потенциала каждому узлу x в лесу непересекающихся множеств после выполнения q операций присваивает потенциал $\phi_q(x)$. Для получения потенциала всего леса мы суммируем потенциалы его узлов: $\Phi_q = \sum_x \phi_q(x)$, где Φ_q обозначает потенциал всего леса после выполнения q операций. До выполнения первой операции лес пуст, и мы полагаем, что $\Phi_0 = 0$. Потенциал Φ_q никогда не может стать отрицательным.

Значение $\phi_q(x)$ зависит от того, является ли x корнем дерева после q-й операции. Если это так или если $x.rank = 0$, то $\phi_q(x) = \alpha(n) \cdot x.rank$.

Теперь предположим, что после q-й операции x не является корнем и что $x.rank \geq 1$. Необходимо определить две вспомогательные функции от x до определения $\phi_q(x)$. Сначала определим

$$\text{level}(x) = \max\{k : x.p.rank \geq A_k(x.rank)\} ,$$

т.е. level(x) — наибольший уровень k, для которого A_k, примененное к рангу x, не превышает ранг родителя x.

Мы утверждаем, что

$$0 \leq \text{level}(x) < \alpha(n) . \tag{21.1}$$

Это можно подтвердить следующим образом. Мы имеем

$$\begin{aligned} x.p.rank &\geq x.rank + 1 && \text{(согласно лемме 21.4)} \\ &= A_0(x.rank) && \text{(согласно определению } A_0(j)) , \end{aligned}$$

откуда вытекает, что level$(x) \geq 0$, и мы имеем

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(поскольку } A_k(j) \text{ строго возрастающая)} \\ &\geq n && \text{(согласно определению } \alpha(n)) \\ &> x.p.rank && \text{(согласно лемме 21.6)} , \end{aligned}$$

откуда вытекает, что level$(x) < \alpha(n)$. Заметим, что, поскольку $x.p.rank$ монотонно растет со временем, то же происходит и с level(x).

Вторая вспомогательная функция применима при $x.rank \geq 1$:

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} ,$$

т.е. функция $\text{iter}(x)$ представляет собой наибольшее количество итеративного применения функции $A_{\text{level}(x)}$ к исходному рангу x, до того как мы получим значение, превышающее ранг родителя x.

Мы утверждаем, что при $x.rank \geq 1$

$$1 \leq \text{iter}(x) \leq x.rank , \tag{21.2}$$

в чем можно убедиться следующим образом. Мы имеем

$$\begin{aligned} x.p.rank &\geq A_{\text{level}(x)}(x.rank) && \text{(согласно определению level}(x)) \\ &= A^{(1)}_{\text{level}(x)}(x.rank) , \end{aligned}$$

где последнее равенство следует из определения функциональной итерации. Отсюда вытекает, что $\text{iter}(x) \geq 1$, и мы имеем

$$\begin{aligned} A^{(x.rank+1)}_{\text{level}(x)}(x.rank) &= A_{\text{level}(x)+1}(x.rank) && \text{(по определению } A_k(j)) \\ &> x.p.rank && \text{(по определению level}(x)) , \end{aligned}$$

откуда следует, что $\text{iter}(x) \leq x.rank$. Заметим, что, поскольку $x.p.rank$ монотонно возрастает со временем, для того, чтобы значение $\text{iter}(x)$ уменьшалось, значение $\text{level}(x)$ должно возрастать. Пока значение $\text{level}(x)$ остается неизменным, значение $\text{iter}(x)$ должно либо увеличиваться, либо вообще не изменяться.

Имея описанные вспомогательные функции, мы можем определить потенциал узла x после q операций:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank , & \text{если } x \text{ является корнем или } x.rank = 0 , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) , & \text{если } x \text{ — не корень и } x.rank \geq 1 . \end{cases}$$

Далее рассмотрим некоторые полезные свойства потенциалов узлов.

Лемма 21.8

Для каждого узла x и для всех количеств операций q имеем

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank .$$

Доказательство. Если x является корнем или $x.rank = 0$, то $\phi_q(x) = \alpha(n) \cdot x.rank$ по определению. Предположим теперь, что x не является корнем и что $x.rank \geq 1$. Мы получим нижнюю границу $\phi_q(x)$ путем максимизации $\text{level}(x)$ и $\text{iter}(x)$. Согласно границе (21.1) $\text{level}(x) \leq \alpha(n) - 1$, а согласно

границе (21.2) $\text{iter}(x) \le x.rank$. Таким образом,

$$\begin{aligned}\phi_q(x) &= (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) \\ &\ge (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - x.rank \\ &= x.rank - x.rank \\ &= 0 .\end{aligned}$$

Аналогично мы получаем верхнюю границу $\phi_q(x)$ минимизацией $\text{level}(x)$ и $\text{iter}(x)$. В соответствии с границей (21.1) $\text{level}(x) \ge 0$, а в соответствии с границей (21.2) $\text{iter}(x) \ge 1$. Таким образом,

$$\begin{aligned}\phi_q(x) &\le (\alpha(n) - 0) \cdot x.rank - 1 \\ &= \alpha(n) \cdot x.rank - 1 \\ &< \alpha(n) \cdot x.rank .\end{aligned}$$

■

Следствие 21.9

Если узел x не является корнем и $x.rank > 0$, то $\phi_q(x) < \alpha(n) \cdot x.rank$. ■

Изменения потенциала и амортизированная стоимость операций

Теперь мы готовы к рассмотрению вопроса о влиянии операций над непересекающимися множествами на потенциалы узлов. Зная, как изменяется потенциал при той или иной операции, мы можем определить амортизированную стоимость каждой операции.

Лемма 21.10

Пусть x — узел, не являющийся корнем, и предположим, что q-я операция — либо LINK, либо FIND-SET. Тогда после выполнения q-й операции $\phi_q(x) \le \phi_{q-1}(x)$. Более того, если $x.rank \ge 1$ и из-за выполнения q-й операции происходит изменение либо $\text{level}(x)$, либо $\text{iter}(x)$, то $\phi_q(x) \le \phi_{q-1}(x) - 1$. То есть потенциал x не может возрастать, и если он имеет положительное значение и либо $\text{level}(x)$, либо $\text{iter}(x)$ изменяется, то потенциал x уменьшается как минимум на 1.

Доказательство. Поскольку x не является корнем, q-я операция не изменяет $x.rank$, и так как n не изменяется после первых n операций MAKE-SET, $\alpha(n)$ остается неизменной величиной. Следовательно, эти компоненты формулы функции потенциала x после выполнения q-й операции не изменяются. Если $x.rank = 0$, то $\phi_q(x) = \phi_{q-1}(x) = 0$. Теперь предположим, что $x.rank \ge 1$.

Вспомним, что значение функции $\text{level}(x)$ монотонно растет со временем. Если q-я операция оставляет значение $\text{level}(x)$ неизменным, то значение $\text{iter}(x)$ либо возрастает, либо остается неизменным. Если и $\text{level}(x)$, и $\text{iter}(x)$ не изменяются, то $\phi_q(x) = \phi_{q-1}(x)$. Если же значение $\text{level}(x)$ не изменяется, а $\text{iter}(x)$ возрастает, то последнее значение увеличивается как минимум на 1, так что $\phi_q(x) \le \phi_{q-1}(x) - 1$.

Наконец, если q-я операция увеличивает $\text{level}(x)$, то это увеличение составляет как минимум 1, так что значение члена $(\alpha(n) - \text{level}(x)) \cdot x.rank$ уменьшается как минимум на величину $x.rank$. Так как значение $\text{level}(x)$ возрастает, значение $\text{iter}(x)$ может уменьшаться, но в соответствии с границей (21.2) это уменьшение не может превышать $x.rank - 1$. Таким образом, увеличение потенциала, вызванное изменением значения $\text{iter}(x)$, меньше, чем уменьшение потенциала из-за изменения $\text{level}(x)$, так что мы можем заключить, что $\phi_q(x) \leq \phi_{q-1}(x) - 1$. ■

Наши последние три леммы показывают, что амортизированная стоимость каждой из операций MAKE-SET, LINK и FIND-SET составляет $O(\alpha(n))$. Вспомним, что согласно (17.2) амортизированная стоимость каждой операции равна ее фактической стоимости плюс увеличение потенциала, вызванное ее выполнением.

Лемма 21.11

Амортизированная стоимость каждой операции MAKE-SET равна $O(1)$.

Доказательство. Предположим, что q-й операцией является MAKE-SET(x). Эта операция создает узел x с рангом 0, так что $\phi_q(x) = 0$. Никакие иные ранги и потенциалы не изменяются, так что $\Phi_q = \Phi_{q-1}$. То, что фактическая стоимость операции MAKE-SET равна $O(1)$, завершает доказательство. ■

Лемма 21.12

Амортизированная стоимость каждой операции LINK равна $O(\alpha(n))$.

Доказательство. Предположим, что q-й операцией является LINK(x, y). Фактическая стоимость операции LINK равна $O(1)$. Без потери общности предположим, что LINK делает y родительским узлом x.

Для определения изменения потенциала из-за выполнения операции LINK заметим, что множество узлов, потенциалы которых могут измениться, ограничено узлами x, y и дочерними узлами y непосредственно перед операцией. Мы покажем, что единственным узлом, потенциал которого может увеличиться в результате выполнения операции LINK, является узел y, и это увеличение не превышает $\alpha(n)$.

- Согласно лемме 21.10 потенциал любого узла, являющегося дочерним узлом y перед выполнением операции LINK, не может увеличиться в результате выполнения этой операции.
- По определению $\phi_q(x)$ мы видим, что, поскольку узел x непосредственно перед q-й операцией был корнем, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$. Если $x.rank = 0$, то $\phi_q(x) = \phi_{q-1}(x) = 0$. В противном случае

$$\begin{aligned} \phi_q(x) &< \alpha(n) \cdot x.rank && \text{(согласно следствию 21.9)} \\ &= \phi_{q-1}(x) , \end{aligned}$$

так что потенциал x уменьшается.

- Поскольку y непосредственно перед выполнением операции LINK был корнем, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. Операция LINK оставляет y корнем и либо оставляет ранг y неизменным, либо увеличивает его на 1. Следовательно, либо $\phi_q(y) = \phi_{q-1}(y)$, либо $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

Таким образом, увеличение потенциала вследствие операции LINK не превышает $\alpha(n)$, и амортизированная стоимость операции LINK равна $O(1) + \alpha(n) = O(\alpha(n))$. ■

Лемма 21.13

Амортизированная стоимость каждой операции FIND-SET равна $O(\alpha(n))$.

Доказательство. Предположим, что q-й операцией является FIND-SET и что путь поиска содержит s узлов. Фактическая стоимость операции FIND-SET составляет $O(s)$. Покажем, что отсутствуют узлы, потенциал которых возрастает вследствие операции FIND-SET, и что как минимум у $\max(0, s - (\alpha(n) + 2))$ узлов на пути поиска потенциал уменьшается по меньшей мере на 1.

Чтобы увидеть отсутствие узлов с возрастающим потенциалом, обратимся к лемме 21.10 для всех узлов, не являющихся корнем. Если же узел x является корнем, то его потенциал равен $\alpha(n) \cdot x.rank$ и остается неизменным.

Теперь покажем, что потенциал как минимум $\max(0, s - (\alpha(n) + 2))$ узлов уменьшается по меньшей мере на 1. Пусть x — узел на пути поиска, такой, что $x.rank > 0$, и за x на пути поиска следует другой узел y, не являющийся корнем, где непосредственно перед выполнением операции FIND-SET $\text{level}(y) = \text{level}(x)$ (узел y не обязательно следует *непосредственно* за x). Этим ограничениям на x удовлетворяют все узлы на пути поиска, кроме не более чем $\alpha(n) + 2$ узлов. Приведенным условиям не удовлетворяют первый узел на пути поиска (если он имеет нулевой ранг), последний узел пути (т.е. корень), а также последний узел w на пути, для которого $\text{level}(w) = k$ для каждого $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Зафиксировав такой узел x, покажем, что потенциал x уменьшается как минимум на 1. Пусть $k = \text{level}(x) = \text{level}(y)$. Непосредственно перед сжатием пути в процедуре FIND-SET мы имеем

$$\begin{aligned}
x.p.rank &\geq A_k^{(\text{iter}(x))}(x.rank) && \text{(по определению iter}(x)) , \\
y.p.rank &\geq A_k(y.rank) && \text{(по определению level}(y)) , \\
y.rank &\geq x.p.rank && \text{(согласно следствию 21.5 и поскольку} \\
& && y \text{ следует за } x \text{ на пути поиска)} .
\end{aligned}$$

Объединив приведенные неравенства и обозначив через i значение $\text{iter}(x)$ перед сжатием пути, получаем

$$\begin{aligned}
y.p.rank &\geq A_k(y.rank) \\
&\geq A_k(x.p.rank) && \text{(поскольку } A_k(j) \text{ строго возрастающая)} \\
&\geq A_k(A_k^{(\text{iter}(x))}(x.rank)) \\
&= A_k^{(i+1)}(x.rank) .
\end{aligned}$$

Поскольку после сжатия пути и x, и y имеют один и тот же родительский узел, мы знаем, что после сжатия пути $x.p.rank = y.p.rank$ и что сжатие пути не уменьшает $y.p.rank$. Поскольку $x.rank$ не изменяется, после сжатия пути $x.p.rank \geq A_k^{(i+1)}(x.rank)$. Таким образом, сжатие пути приводит к тому, что либо увеличивается $\text{iter}(x)$ (как минимум до $i + 1$), либо увеличивается $\text{level}(x)$ (что происходит, когда $\text{iter}(x)$ увеличивается как минимум до $x.rank + 1$). В любом случае в соответствии с леммой 21.10 $\phi_q(x) \leq \phi_{q-1}(x) - 1$. Следовательно, потенциал x уменьшается как минимум на 1.

Амортизированная стоимость операции FIND-SET равна фактической стоимости плюс изменение потенциала. Фактическая стоимость равна $O(s)$, и мы показали, что общий потенциал уменьшается как минимум на $\max(0, s - (\alpha(n) + 2))$. Амортизированная стоимость, следовательно, не превышает $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, так как мы можем масштабировать потенциал таким образом, чтобы можно было пренебречь константой, скрытой в $O(s)$. ■

Теперь, после того как мы доказали все приведенные выше леммы, перейдем к следующей теореме.

Теорема 21.14

Последовательность из m операций MAKE-SET, UNION и FIND-SET, n из которых — операции MAKE-SET, может быть выполнена над лесом непересекающихся множеств с использованием объединения по рангу и сжатия пути за время $O(m\,\alpha(n))$ в наихудшем случае.

Доказательство. Непосредственно следует из лемм 21.7 и 21.11–21.13. ■

Упражнения

21.4.1

Докажите лемму 21.4.

21.4.2

Докажите, что каждый узел имеет ранг, не превышающий $\lfloor \lg n \rfloor$.

21.4.3

Сколько в свете упр. 21.4.2 битов требуется для хранения $x.rank$ для каждого узла x?

21.4.4

Используя решение упр. 21.4.2, приведите простое доказательство того факта, что операции над лесом непересекающихся множеств с использованием объединения по рангу, но без сжатия пути, выполняются за время $O(m \lg n)$.

21.4.5

Профессор полагает, что поскольку ранг узла строго возрастает вдоль простого пути к корню, уровни узлов должны монотонно возрастать вдоль этого пути.

Другими словами, если $x.rank > 0$ и $x.p$ — не корень, то $\text{level}(x) \leq \text{level}(x.p)$. Прав ли профессор?

21.4.6 ★

Рассмотрим функцию $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$. Покажите, что для всех практических значений n выполняется неравенство $\alpha'(n) \leq 3$. Покажите также, воспользовавшись решением упр. 21.4.2, каким образом следует модифицировать функцию потенциала для доказательства того, что последовательность из m операций MAKE-SET, UNION и FIND-SET, n из которых — операции MAKE-SET, может быть выполнена над лесом непересекающихся множеств с использованием объединения по рангу и сжатия пути за время $O(m\,\alpha'(n))$ в наихудшем случае.

Задачи

21.1. Минимум в автономном режиме

Задача ***поиска минимума в автономном режиме*** (off-line minimum problem) состоит в следующем. Пусть имеется динамическое множество T с элементами из области определения $\{1, 2, \ldots, n\}$, поддерживающее операции INSERT и EXTRACT-MIN. Задана последовательность S из n вызовов INSERT и m вызовов EXTRACT-MIN, в которой каждый ключ из $\{1, 2, \ldots, n\}$ вставляется ровно один раз. Мы хотим определить, какой ключ возвращается каждым вызовом EXTRACT-MIN, т.е. заполнить массив $extracted[1\,..\,m]$, где $extracted[i]$ (для $i = 1, 2, \ldots, m$) представляет собой ключ, возвращаемый i-м вызовом EXTRACT-MIN. Задача "автономна" в том смысле, что перед тем как приступить к вычислениям, ей известна вся последовательность S.

а. В следующем экземпляре задачи каждый вызов INSERT(i) представлен вставляемым числом i, а каждый вызов EXTRACT-MIN представлен буквой E:

$$4, 8, \text{E}, 3, \text{E}, 9, 2, 6, \text{E}, \text{E}, \text{E}, 1, 7, \text{E}, 5\ .$$

Заполните массив *extracted* корректными значениями.

Чтобы разработать алгоритм для решения этой задачи, разобьем последовательность S на гомогенные подпоследовательности, т.е. представим S в таком виде:

$$\text{I}_1, \text{E}, \text{I}_2, \text{E}, \text{I}_3, \ldots, \text{I}_m, \text{E}, \text{I}_{m+1}\ ,$$

где каждая буква E представляет один вызов EXTRACT-MIN, а I_j представляет (возможно, пустую) последовательность вызовов INSERT. Для каждой подпоследовательности I_j мы сначала помещаем ключи, вставленные данными операциями в множество K_j (которое является пустым, если подпоследовательность I_j пуста). Затем делаем следующее.

```
OFF-LINE-MINIMUM(m, n)
for i = 1 to n
    Определяем j, такое, что i ∈ K_j
    if j ≠ m + 1
        extracted[j] = i
        Пусть l — наименьшее значение, большее, чем j,
            для которого существует множество K_l
        K_l = K_j ∪ K_l, с уничтожением K_j
return extracted
```

б. Докажите корректность массива *extracted*, возвращаемого процедурой OFF-LINE-MINIMUM.

в. Опишите, как эффективно реализовать процедуру OFF-LINE-MINIMUM с использованием структур данных для непересекающихся множеств. Дайте точную оценку времени работы вашей реализации в наихудшем случае.

21.2. Определение глубины

В ***задаче по определению глубины*** (depth-determination problem) мы работаем с лесом $\mathcal{F} = \{T_i\}$ корневых деревьев, в котором поддерживаются следующие три операции.

MAKE-TREE(v) создает дерево, состоящее из единственного узла v.

FIND-DEPTH(v) возвращает глубину узла v в его дереве.

GRAFT(r, v) ("прививка") делает узел r, являющийся корнем, дочерним по отношению к узлу v, который находится в дереве, отличном от дерева r, но при этом не обязательно должен сам быть корнем.

а. Предположим, что мы используем представление дерева, аналогичное лесу непересекающихся множеств: $v.p$ является родительским узлом v, и если v — корень, то $v.p = v$. Реализуем процедуру GRAFT(r, v) путем присваивания $r.p = v$, а FIND-DEPTH(v) — как следование по пути поиска к корню с подсчетом всех узлов (кроме v), встречающихся на этом пути. Покажите, что время работы последовательности из m процедур MAKE-TREE, FIND-DEPTH и GRAFT в наихудшем случае равно $\Theta(m^2)$.

Используя объединение по рангу и сжатие пути, мы можем сократить время работы в наихудшем случае. Воспользуемся лесом непересекающихся множеств $\mathcal{S} = \{S_i\}$, где каждое множество S_i (представляющее собой дерево) соответствует дереву T_i в лесу $\mathcal{F}$. Структура дерева в S_i не обязательно соответствует структуре дерева T_i. Однако хотя реализация S_i не хранит точные отношения "родитель–потомок", она позволяет определить глубину любого узла в T_i.

Ключевая идея состоит в хранении в каждом узле v "псевдодистанции" $v.d$, которая определена таким образом, что сумма псевдодистанций вдоль простого пути от v к корню его множества S_i равна глубине v в T_i. Иначе говоря, если простой путь от v к корню в S_i представляет собой последовательность $v_0, v_1, \ldots, v_k$, где $v_0 = v$, а v_k — корень S_i, то глубина v в T_i равна $\sum_{j=0}^{k} v_j.d$.

б. Приведите реализацию процедуры MAKE-TREE.

в. Покажите, как следует изменить процедуру FIND-SET для реализации процедуры FIND-DEPTH. Ваша реализация должна осуществлять сжатие пути, а время ее работы должно линейно зависеть от длины пути поиска. Убедитесь в корректности обновления псевдодистанций вашей процедурой.

г. Покажите, как можно реализовать процедуру GRAFT(r, v), которая объединяет множества, содержащие r и v, модифицируя процедуры UNION и LINK. Убедитесь в корректности обновления псевдодистанций вашей процедурой. Не забывайте, что корень множества S_i не обязательно должен быть корнем соответствующего дерева T_i.

д. Дайте точную оценку времени работы в наихудшем случае для последовательности из m операций MAKE-TREE, FIND-DEPTH и GRAFT, n из которых — операции MAKE-TREE.

21.3. Алгоритм Таржана для автономного поиска наименьшего общего предка

Наименьший общий предок (least common ancestor) двух узлов u и v в корневом дереве T представляет собой узел w, который среди узлов, являющихся предками как u, так и v, имеет наибольшую глубину. В ***задаче автономного поиска наименьшего общего предка*** (off-line least-common-ancestor problem) даны корневое дерево T и произвольное множество $P = \{\{u, v\}\}$ неупорядоченных пар узлов в T. Для каждой пары узлов из P требуется найти их наименьший общий предок.

Для решения задачи автономного поиска наименьшего общего предка осуществляется обход дерева T с помощью приведенной ниже процедуры с начальным вызовом LCA($T.root$). Предполагается, что перед вызовом процедуры все узлы помечены как WHITE (белые).

```
LCA(u)
 1  MAKE-SET(u)
 2  FIND-SET(u).ancestor = u
 3  for каждого узла v, дочернего по отношению к u в T
 4      LCA(v)
 5      UNION(u, v)
 6      FIND-SET(u).ancestor = u
 7  u.color = BLACK
 8  for каждого узла v, такого, что {u, v} ∈ P
 9      if v.color == BLACK
10          print "Наименьшим общим предком"
                u "и" v "является" FIND-SET(v).ancestor
```

а. Докажите, что строка 10 выполняется только один раз для каждой пары $\{u, v\} \in P$.

б. Докажите, что в момент вызова LCA(u) количество множеств в структуре данных для непересекающихся множеств равно глубине u в T.

в. Докажите, что процедура LCA правильно выводит наименьшие общие предки u и v для каждой пары $\{u, v\} \in P$.

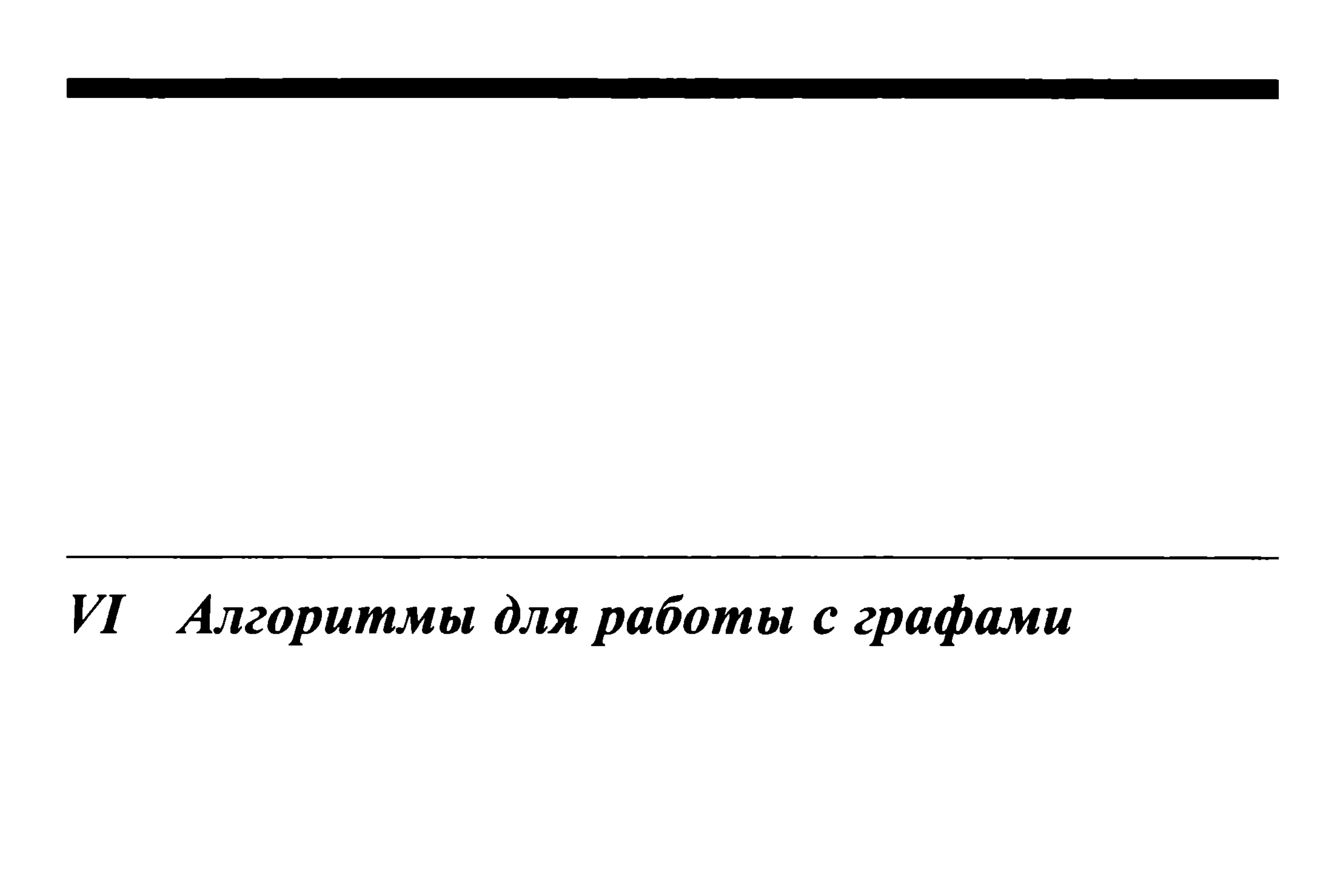
г. Проанализируйте время работы процедуры LCA в предположении, что используется реализация структуры непересекающихся множеств из раздела 21.3.

Заключительные замечания

Многие важные научные результаты о структурах данных для непересекающихся множеств в той или иной степени принадлежат Р.Э. Таржану (R.E. Tarjan). В частности, именно им [326, 328] дана первая точная верхняя граница с применением очень медленно растущей инверсии $\widehat{\alpha}(m, n)$ функции Аккермана (Ackermann). (Функция $A_k(j)$, используемая в разделе 21.4, подобна функции Аккермана, а функция $\alpha(n)$ — обратной ей. Для всех мыслимых значений m и n как $\alpha(n)$, так и $\widehat{\alpha}(m, n)$ не превышают 4.) Верхняя граница $O(m \lg^* n)$ была доказана несколько ранее Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5, 178]. Материал раздела 21.4 основан на более позднем анализе Таржана [330], который, в свою очередь, опирается на анализ Козена (Kozen) [219]. Харфст (Harfst) и Рейнгольд (Reingold) [160] разработали версию доказательства полученных Таржаном оценок с помощью потенциалов.

Таржан и ван Леувен (van Leeuwen) [331] рассмотрели разные варианты эвристики со сжатием пути, включая однопроходные варианты, которые эффективнее двухпроходных в силу меньшего постоянного множителя. Позже Харфст и Рейнгольд [160] показали, какие небольшие изменения следует внести в потенциальную функцию для адаптации их анализа сжатия пути к однопроходному варианту Таржана. Габов (Gabow) и Таржан [120] показали, что в некоторых приложениях операции над непересекающимися множествами могут выполняться за время $O(m)$.

Таржан [327] показал, что для операций над произвольными структурами данных для непересекающихся множеств, удовлетворяющими определенным техническим условиям, нижняя граница времени работы равна $\Omega(m\, \widehat{\alpha}(m, n))$. Позже эта нижняя граница была обобщена Фредманом (Fredman) и Саксом (Saks) [112], которые показали, что в наихудшем случае эти операции требуют обращения к $\Omega(m\, \widehat{\alpha}(m, n))$ ($\lg n$)-битовым словам памяти.



VI Алгоритмы для работы с графами

Введение

Графы представляют собой распространенные структуры в информатике, и алгоритмы для работы с графами очень важны. Имеются сотни интересных вычислительных задач, сформулированных с использованием графов. В этой части мы коснемся только некоторых наиболее важных из них.

В главе 22 рассматриваются вопросы представления графов в компьютере и обсуждаются алгоритмы, основанные на обходе графа в ширину и в глубину. Здесь приведены два применения обхода в глубину — топологическая сортировка ориентированного ациклического графа и разложение ориентированного графа на сильно связные компоненты.

В главе 23 описывается вычисление остовного дерева минимального веса. Такое дерево представляет собой набор ребер, связывающий все вершины графа с наименьшим возможным весом (каждое ребро обладает некоторым весом). Эта задача служит хорошим примером применения жадных алгоритмов (см. главу 16).

В главах 24 и 25 рассматривается задача вычисления кратчайшего пути между вершинами, когда каждому ребру присвоена некоторая длина или вес. Глава 24 посвящена вычислению кратчайшего пути из одной вершины во все остальные, а в главе 25 рассматривается поиск кратчайших путей для всех пар вершин.

И наконец в главе 26 рассматривается задача о вычислении максимального потока материала в сети (ориентированном графе) с определенным источником вещества, стоком и пропускной способностью каждого ребра. К этой общей задаче сводятся многие другие, так что хороший алгоритм ее решения может использоваться во многих приложениях.

При описании времени работы алгоритма над заданным графом $G = (V, E)$ мы обычно определяем размер входного графа в терминах количества его вершин $|V|$ и ребер $|E|$, т.е. размер входных данных описывается двумя, а не одним параметром. Для удобства и краткости в асимптотических обозначениях (таких, как O и Θ-обозначения), и *только* в них, символ V будет означать $|V|$, а символ E — $|E|$, т.е. когда мы будем говорить "время работы алгоритма равно $O(VE)$", то

это означает "время работы алгоритма равно $O(|V|\,|E|)$". Такое соглашение позволяет сделать формулы для времени работы более удобочитаемыми без риска неоднозначного толкования.

Еще одно соглашение принято для псевдокода. Мы обозначаем множество вершин графа G как $G.V$, а множество ребер — как $G.E$, т.е. в псевдокоде множества вершин и ребер рассматриваются как атрибуты графа.

Глава 22. Элементарные алгоритмы для работы с графами

В этой главе рассматриваются методы представления графов, а также их обхода. Под обходом графа понимается систематическое перемещение по ребрам графа, при котором посещаются все его вершины. Алгоритм обхода графа может многое сказать о его структуре, поэтому многие алгоритмы начинают свою работу с получения информации о структуре путем обхода графа. Некоторые другие алгоритмы для работы с графами организованы как простое усовершенствование базовых алгоритмов обхода графа.

В разделе 22.1 рассматриваются два наиболее распространенных способа представления графов — списки смежных вершин и матрица смежности. В разделе 22.2 представлены простой алгоритм обхода графа, который называется поиском в ширину, и соответствующее этому алгоритму дерево. В разделе 22.3 представлен алгоритм поиска в глубину и доказываются некоторые свойства этого вида обхода графа. В разделе 22.4 вы познакомитесь с первым реальным применением поиска в глубину — топологической сортировкой ориентированного ациклического графа. Еще одно применение поиска в глубину — поиск сильно связных компонентов графа — приводится в разделе 22.5.

22.1. Представление графов

Имеется два стандартных способа представления графа $G = (V, E)$: как набора списков смежных вершин и как матрицы смежности. Оба способа представления применимы как для ориентированных, так и для неориентированных графов. Обычно более предпочтительным является представление с помощью списков смежности, поскольку оно обеспечивает компактное представление ***разреженных*** (sparse) графов, т.е. таких, для которых $|E|$ гораздо меньше, чем $|V|^2$. Большинство алгоритмов, представленных в данной книге, предполагают, что входной граф представлен именно в виде списка смежности. Представление с помощью матрицы смежности предпочтительнее в случае ***плотных*** (dense) графов (т.е. когда значение $|E|$ близкó к $|V|^2$) или когда необходимо иметь возможность быстро определить, существует ли ребро, соединяющие две данные вершины. Например, в главе 25 два алгоритма поиска кратчайшего пути для всех пар вершин используют представление графов именно в виде матриц смежности.

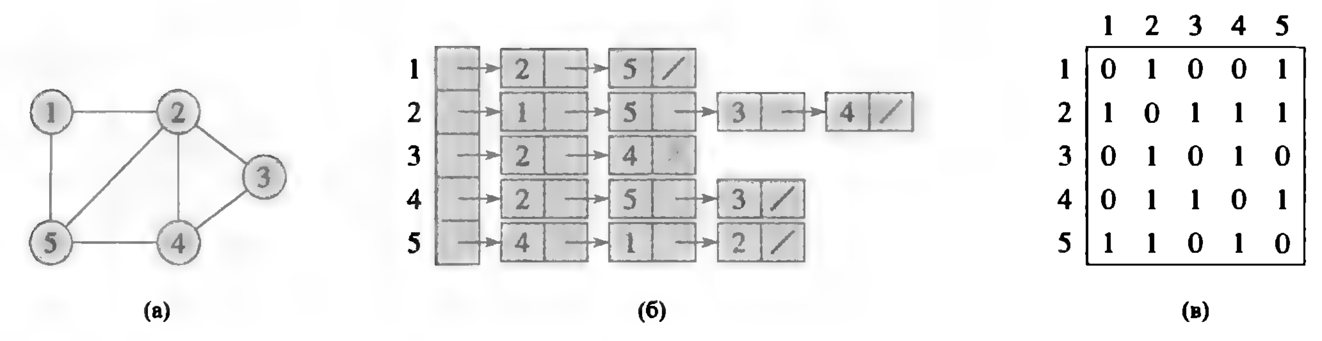


Рис. 22.1. Два представления неориентированного графа. **(а)** Неориентированный граф G с пятью вершинами и семью ребрами. **(б)** Представление G с помощью списков смежности. **(в)** Представление G с помощью матрицы смежности.

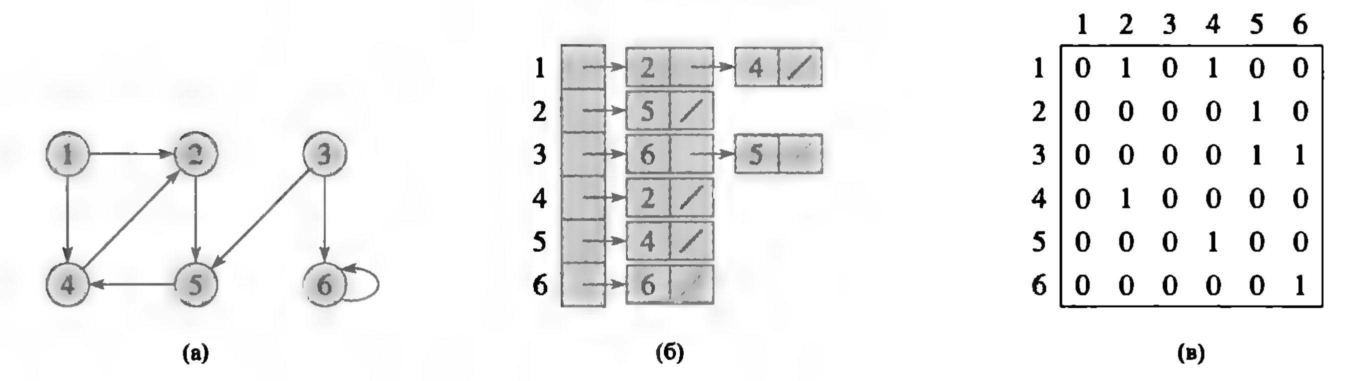


Рис. 22.2. Два представления ориентированного графа. **(а)** Ориентированный граф G с шестью вершинами и восемью ребрами. **(б)** Представление G с помощью списков смежности. **(в)** Представление G с помощью матрицы смежности.

Представление графа $G = (V, E)$ в виде ***списка смежности*** (adjacency-list representation) использует массив Adj из $|V|$ списков, по одному для каждой вершины из V. Для каждой вершины $u \in V$ список смежности $Adj[u]$ содержит все вершины v, такие, что $(u, v) \in E$, т.е. $Adj[u]$ состоит из всех вершин, смежных с u в графе G (список может содержать и не сами вершины, а указатели на них). Поскольку списки смежности представляют ребра графа, в псевдокоде массив Adj рассматривается как атрибут графа, так же, как мы рассматриваем множество ребер E. Таким образом, в псевдокоде вы встретитесь с обозначениями наподобие $G.Adj[u]$. На рис. 22.1, (б) показано представление неориентированного графа на рис. 22.1, (а) с помощью списков смежности. Аналогично на рис. 22.2, (б) показано представление с помощью списков смежности ориентированного графа на рис. 22.2, (а).

Если G является ориентированным графом, то сумма длин всех списков смежности равна $|E|$, поскольку ребру (u, v) однозначно соответствует элемент v в списке $Adj[u]$. Если же G представляет собой неориентированный граф, то сумма длин всех списков смежности равна $2|E|$, поскольку ребро (u, v), будучи неориентированным, появляется как в списке смежности u, так и в списке v. Как для ориентированных, так и для неориентированных графов представление в виде списков требует объема памяти, равного $\Theta(V + E)$.

Списки смежности легко адаптируются для представления ***взвешенных графов*** (weighted graph), т.е. графов, с каждым ребром которых связан определенный ***вес***

(weight), обычно определяемый ***весовой функцией*** (weight function) $w : E \to \mathbb{R}$. Например, пусть $G = (V, E)$ — взвешенный граф с весовой функцией w. Вес $w(u, v)$ ребра $(u, v) \in E$ просто хранится вместе с вершиной v в списке смежности u. Представление с помощью списков смежности достаточно гибко в том смысле, что легко адаптируется для поддержки многих других вариантов графов.

Потенциальный недостаток представления с помощью списков смежности заключается в том, что при этом нет более быстрого способа определить, имеется ли данное ребро (u, v) в графе, чем поиск v в списке $Adj[u]$. Этот недостаток можно устранить ценой использования асимптотически большего количества памяти в представлении графа с помощью матрицы смежности. (В упр. 22.1.8 предлагается вариант списков смежности, позволяющий ускорить поиск ребер.)

Представление графа $G = (V, E)$ с помощью ***матрицы смежности*** (adjacency-matrix representation) предполагает, что вершины графа перенумерованы в некотором произвольном порядке числами $1, 2, \ldots, |V|$. В таком случае представление графа G с использованием матрицы смежности представляет собой матрицу $A = (a_{ij})$ размером $|V| \times |V|$, такую, что

$$a_{ij} = \begin{cases} 1 \ , & \text{если } (i, j) \in E \ , \\ 0 & \text{в противном случае} \ . \end{cases}$$

На рис. 22.1, (в) и 22.2, (в) показаны матрицы смежности неориентированного графа и ориентированного графа, изображенных на рис. 22.1, (а) и 22.2, (а) соответственно. Матрице смежности графа требуется $\Theta(V^2)$ памяти независимо от количества ребер графа.

Обратите внимание на симметричность матрицы смежности на рис. 22.1, (в) относительно главной диагонали. Поскольку граф неориентирован, (u, v) и (v, u) представляют одно и то же ребро, так что матрица смежности неориентированного графа совпадает с транспонированной матрицей смежности, т.е. $A = A^{\mathrm{T}}$. В ряде приложений это свойство позволяет хранить только элементы матрицы, находящиеся на главной диагонали и выше нее, что позволяет почти в два раза сократить необходимое количество памяти.

Так же, как и представление со списками смежности, представление с матрицами смежности можно использовать для взвешенных графов. Например, если $G = (V, E)$ — взвешенный граф с весовой функцией w, то вес $w(u, v)$ ребра $(u, v) \in E$ хранится в записи в строке u и столбце v матрицы смежности. Если ребро не существует, то в соответствующем элементе матрицы хранится значение NIL, хотя для многих приложений удобнее использовать некоторое значение, такое как 0 или ∞.

Хотя список смежности асимптотически как минимум столь же эффективен в плане требуемой памяти, как и матрица смежности, простота последней делает ее предпочтительной при работе с небольшими графами. Кроме того, в случае невзвешенных графов для представления одного ребра достаточно одного бита, что позволяет существенно сэкономить необходимую для представления память.

Представление атрибутов

Большинству алгоритмов для работы с графами требуется поддержка атрибутов для вершин и/или ребер. Мы указываем эти атрибуты с помощью обычных обозначений, таких как $v.d$ в случае атрибута d вершины v. Когда мы указываем ребра как пары вершин, то используем тот же стиль обозначений. Например, если ребра имеют атрибут f, то этот атрибут для ребра (u, v) обозначается как $(u, v).f$. Для представления и понимания алгоритмов такой записи атрибутов вполне достаточно.

Реализация атрибутов вершин и ребер в реальных программах — это совершенно другая история. Нет некоторого наилучшего способа хранения атрибутов вершин и ребер и работы с ними. В каждой конкретной ситуации решение, скорее всего, будет зависеть от используемого языка программирования, реализуемого алгоритма и от того, как остальная часть программы использует граф. Если граф представлен с помощью списков смежности, один из вариантов представления атрибутов — применение дополнительных массивов, таких как массив $d[1..|V|]$, параллельных массиву Adj. Если вершины, смежные с u, находятся в $Adj[u]$, то то, что мы называем атрибутом $u.d$, в действительности хранится в элементе массива $d[u]$. Возможны и многие иные способы реализации атрибутов. Например, в объектно-ориентированном языке программирования атрибуты вершин могут быть представлены как переменные экземпляров в подклассе класса `Vertex`.

Упражнения

22.1.1

Имеется представление ориентированного графа с использованием списков смежности. Как долго будут вычисляться исходящие степени всех вершин графа? А входящие степени?

22.1.2

Имеется представление с использованием списков смежности полного бинарного дерева с 7 вершинами. Приведите его представление с использованием матрицы смежности (считаем, что вершины пронумерованы от 1 до 7, как в бинарной пирамиде).

22.1.3

При ***транспонировании*** (transpose) ориентированного графа $G = (V, E)$ мы получаем граф $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, где $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$, т.е. граф G^{T} представляет собой граф G с обратным направлением ребер. Опишите эффективный алгоритм транспонирования графа как для представления с использованием списков смежности, так и для матриц смежности. Проанализируйте время работы своих алгоритмов.

22.1.4

Имеется представление мультиграфа $G = (V, E)$ с использованием списков смежности. Опишите алгоритм со временем работы $O(V + E)$ для вычисления представления со списками смежности "эквивалентного" неориентированного графа

$G' = (V, E')$, где E' состоит из ребер из E, где кратные ребра заменены обычными и удалены ребра-циклы.

22.1.5

Квадратом (square) ориентированного графа $G = (V, E)$ является граф $G^2 = (V, E^2)$, такой, что $(u, v) \in E^2$ тогда и только тогда, когда G содержит путь между u и v, состоящий не более чем из двух ребер. Опишите эффективный алгоритм вычисления квадрата графа как для представления с использованием списков смежности, так и для матриц смежности. Проанализируйте время работы своих алгоритмов.

22.1.6

При использовании матриц смежности большинство алгоритмов для работы с графами требуют времени $\Omega(V^2)$, но имеются и некоторые исключения. Покажите, что определить, содержит ли граф G ***всеобщий сток*** (universal sink), т.е. вершину с входящей степенью, равной $|V| - 1$, и с исходящей степенью 0, можно за время $O(V)$, если использовать представление графа с помощью матрицы смежности.

22.1.7

Матрицей инцидентности (incidence matrix) ориентированного графа без петель $G = (V, E)$ является матрица $B = (b_{ij})$ размером $|V| \times |E|$, такая, что

$$b_{ij} = \begin{cases} -1 , & \text{если ребро } j \text{ выходит из вершины } i, \\ 1 , & \text{если ребро } j \text{ входит в вершину } i, \\ 0 & \text{в противном случае} . \end{cases}$$

Поясните, что представляют собой элементы матричного произведения BB^{T}, где B^{T} — транспонированная матрица B.

22.1.8

Предположим, что вместо связанного списка каждый элемент массива $Adj[u]$ представляет собой хеш-таблицу, содержащую вершины v, для которых $(u, v) \in E$. Чему равно ожидаемое время определения наличия ребра в графе, если проверка всех ребер выполняется с одинаковой вероятностью. Какие недостатки имеет данная схема? Предложите другие структуры данных для списков ребер, которые позволяют решать поставленную задачу. Имеет ли ваша схема преимущества или недостатки по сравнению с хеш-таблицами?

22.2. Поиск в ширину

Поиск в ширину (breadth-first search) представляет собой один из простейших алгоритмов для обхода графа и является основой для многих важных алгоритмов для работы с графами. Например, алгоритм Прима (Prim) поиска минимального

остовного дерева (раздел 23.2) или алгоритм Дейкстры (Dijkstra) поиска кратчайших путей из одной вершины (раздел 24.3) применяют идеи, сходные с идеями, используемыми при поиске в ширину.

Пусть задан граф $G = (V, E)$ и выделена ***исходная*** вершина (источник, source) s. Алгоритм поиска в ширину систематически обходит все ребра G для "открытия" всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество ребер) от s до каждой достижимой из s вершины. Кроме того, в процессе обхода строится "дерево поиска в ширину" с корнем s, содержащее все достижимые вершины. Для каждой достижимой из s вершины v простой путь в дереве поиска в ширину соответствует "кратчайшему (т.е. содержащему наименьшее количество ребер) пути" от s до v в G, т.е. пути, содержащему наименьшее количество ребер. Алгоритм работает как для ориентированных, так и для неориентированных графов.

Поиск в ширину имеет такое название потому, что в процессе обхода мы идем вширь, т.е. перед тем как приступить к поиску вершин на расстоянии $k + 1$, выполняется обход всех вершин на расстоянии k.

Для отслеживания работы алгоритма поиск в ширину раскрашивает вершины графа в белый, серый и черный цвета. Изначально все вершины белые, и позже они могут стать серыми, а затем черными. Когда вершина впервые ***открывается*** (discovered) в процессе поиска, она окрашивается. Таким образом, серые и черные вершины — это вершины, которые уже были открыты, но алгоритм поиска в ширину по-разному работает с ними, чтобы обеспечить заявленный порядок обхода.[1] Если $(u, v) \in E$ и вершина u черного цвета, то вершина v либо серая, либо черная, т.е. все вершины, смежные с черной, уже открыты. Серые вершины могут иметь белых соседей, представляя собой границу между открытыми и неоткрытыми вершинами.

Поиск в ширину строит дерево поиска в ширину, которое изначально состоит из одного корня, которым является исходная вершина s. Если в процессе сканирования списка смежности уже открытой вершины u открывается белая вершина v, то вершина v и ребро (u, v) добавляются в дерево. Мы говорим, что u является ***предшественником*** (predecessor), или ***родителем*** (parent), v в дереве поиска вширь. Поскольку вершина может быть открыта не более одного раза, она имеет не более одного родителя. Взаимоотношения предков и потомков определяются в дереве поиска в ширину как обычно: если u находится на простом пути от корня s к вершине v, то u является предком v, а v — потомком u.

Приведенная ниже процедура поиска в ширину BFS предполагает, что входной граф $G = (V, E)$ представлен с помощью списков смежности. Кроме того, поддерживаются дополнительные атрибуты в каждой вершине графа. Цвет каждой вершины $u \in V$ хранится в атрибуте $u.color$, а предшественник — в атрибуте $u.\pi$. Если предшественника у u нет (например, если $u = s$ или u не открыта),

---

[1] Различие между серыми и черными вершинами позволяет легче разобраться в работе алгоритма поиска в ширину. В действительности, как показано в упр. 22.2.3, тот же результат можно получить, если не различать серые и черные вершины.

то $u.\pi = \text{NIL}$. Расстояние от s до вершины u, вычисляемое алгоритмом, хранится в атрибуте $u.d$. Для работы с множеством серых вершин алгоритм использует очередь Q (см. раздел 10.1).

```
BFS(G, s)
1  for Каждой вершины u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for Каждой вершины v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

На рис. 22.3 продемонстрировано применение процедуры BFS.

Процедура BFS работает следующим образом. В строках 1–4 все вершины, за исключением исходной вершины s, окрашиваются в белый цвет, для каждой вершины u атрибуту $u.d$ присваивается значение ∞, а в качестве родителя для каждой вершины устанавливается NIL. В строке 5 исходная вершина s окрашивается в серый цвет, поскольку она рассматривается как открытая в начале процедуры. В строке 6 ее атрибуту $s.d$ присваивается значение 0, а в строке 7 ее родителем становится NIL. В строках 8 и 9 создается пустая очередь Q, в которую помещается единственная вершина s.

Цикл **while** в строках 10–18 выполняется до тех пор, пока остаются серые вершины (т.е. открытые вершины, списки смежности которых еще не просмотрены). Инвариант данного цикла выглядит следующим образом.

> При выполнении проверки в строке 10 очередь Q состоит из множества серых вершин.

Хотя мы не намерены использовать этот инвариант для доказательства корректности алгоритма, легко увидеть, что он выполняется перед первой итерацией и что каждая итерация цикла сохраняет инвариант. Перед первой итерацией единственной серой вершиной и единственной вершиной в очереди Q является исходная вершина s. В строке 11 определяется серая вершина u в голове очереди Q, которая затем удаляется из очереди. Цикл **for** в строках 12–17 просматривает все вершины v в списке смежности u. Если вершина v белая, значит, она еще не открыта.

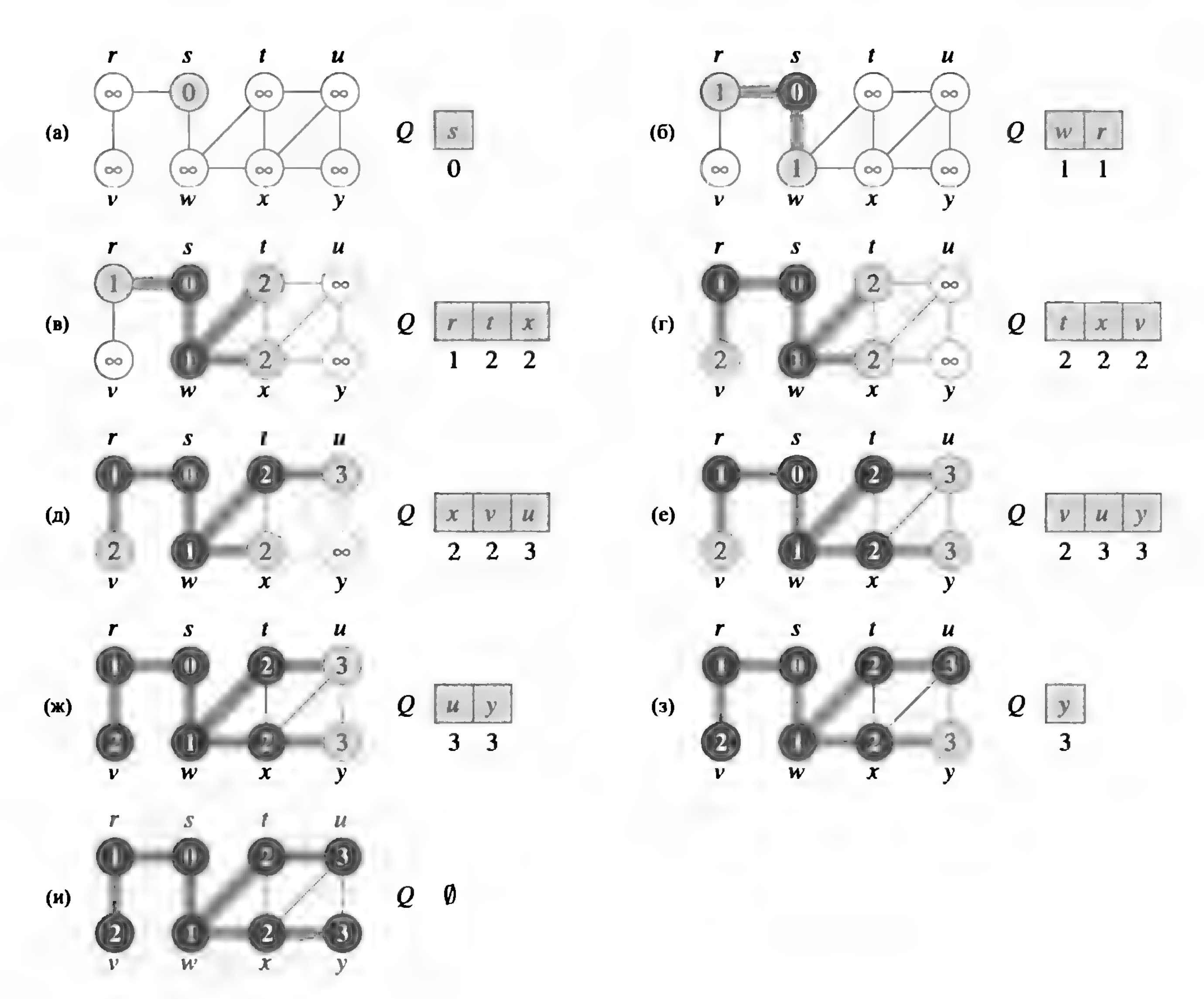


Рис. 22.3. Выполнение процедуры BFS для неориентированного графа. Ребра дерева, добавляемые процедурой BFS, заштрихованы. В каждой вершине u показано соответствующее значение атрибута $u.d$. Состояние очереди Q показано на момент начала каждой итерации цикла **while** в строках 10–18. В очереди под вершинами указаны расстояния до них.

и алгоритм открывает ее, выполняя строки 14–17. Вершине назначается серый цвет, дистанция $v.d$ устанавливается равной $u.d + 1$, а в качестве ее родителя $v.\pi$ указывается вершина u. После этого вершина помещается в хвост очереди Q. После того как все вершины из списка смежности u просмотрены, вершина u окрашивается в строке 18 в черный цвет. Инвариант цикла сохраняется, так как все вершины, которые окрашиваются в серый цвет (строка 14), вносятся в очередь (строка 17), а вершина, которая удаляется из очереди (строка 11), окрашивается в черный цвет (строка 18).

Результат поиска в ширину может зависеть от порядка просмотра вершин, смежных с данной вершиной, в строке 12. Дерево поиска в ширину при этом может варьироваться, но расстояния d, вычисленные алгоритмом, от порядка просмотра не зависят (см. упр. 22.2.5).

Анализ

Перед тем как рассматривать различные свойства поиска в ширину, начнем с самого простого — оценки времени работы алгоритма для входного графа $G = (V, E)$. Мы воспользуемся групповым анализом, описанным в разделе 17.1. После инициализации ни одна вершина не окрашивается в белый цвет, так что проверка в строке 13 гарантирует, что каждая вершина вносится в очередь не более одного раза, а следовательно, и удаляется из очереди она не более одного раза. Операции внесения в очередь и удаления из нее требуют времени $O(1)$, поэтому общее время операций с очередью составляет $O(V)$. Поскольку каждый список смежности сканируется только при удалении соответствующей вершины из очереди, каждый список сканируется не более одного раза. Так как сумма длин всех списков смежности равна $\Theta(E)$, общее время, необходимое для сканирования списков, равно $O(E)$. Накладные расходы на инициализацию равны $O(V)$, так что общее время работы процедуры BFS составляет $O(V + E)$. Таким образом, время поиска в ширину линейно зависит от размера представления графа G с использованием списков смежности.

Кратчайшие пути

В начале этого раздела мы говорили о том, что поиск в ширину находит расстояния до каждой достижимой вершины в графе $G = (V, E)$ от исходной вершины $s \in V$. Определим ***длину кратчайшего пути*** (shortest-path distance) $\delta(s, v)$ от s до v как минимальное количество ребер на любом из путей от s к v. Если пути от s к v не существует, то $\delta(s, v) = \infty$. Путь длиной $\delta(s, v)$ от s к v называется ***кратчайшим путем***[2] (shortest path) от s к v. Перед тем как показать, что поиск в ширину вычисляет длины кратчайших путей, рассмотрим важное свойство длин кратчайших путей.

Лемма 22.1

Пусть $G = (V, E)$ является ориентированным или неориентированным графом и пусть $s \in V$ — произвольная вершина. Тогда для любого ребра $(u, v) \in E$

$$\delta(s, v) \leq \delta(s, u) + 1 .$$

Доказательство. Если вершина u достижима из s, то достижима и вершина v. В этом случае кратчайший путь из s в v не может быть длиннее, чем кратчайший путь из s в u, за которым следует ребро (u, v), так что указанное неравенство выполняется. Если же вершина u недостижима из вершины s, то $\delta(s, u) = \infty$ и неравенство выполняется и в этом случае. ■

---

[2]В главах 24 и 25 мы обобщим понятие кратчайшего пути для взвешенных графов, в которых каждое ребро имеет вес, выраженный действительным числом, а вес пути равен весу составляющих его ребер. Графы, рассматриваемые в данной главе, являются невзвешенными (или, что то же самое, все ребра имеют единичный вес).

Мы хотим показать, что процедура BFS корректно вычисляет $v.d = \delta(s, v)$ для каждой вершины $v \in V$. Сначала покажем, что $v.d$ ограничивает $\delta(s, v)$ сверху.

Лемма 22.2

Пусть $G = (V, E)$ является ориентированным или неориентированным графом и пусть процедура BFS выполняется над ним с исходной вершиной $s \in V$. Тогда по завершении процедуры для каждой вершины $v \in V$ значение $v.d$, вычисленное процедурой BFS, удовлетворяет неравенству $v.d \geq \delta(s, v)$.

Доказательство. Используем индукцию по количеству операций ENQUEUE. Наша гипотеза индукции заключается в том, что для всех $v \in V$ выполняется условие $v.d \geq \delta(s, v)$.

Базисом индукции является ситуация непосредственно после внесения s в очередь в строке 9 процедуры BFS. В этой ситуации гипотеза индукции справедлива, поскольку $s.d = 0 = \delta(s, s)$, а для всех $v \in V - \{s\}$ выполняется $v.d = \infty \geq \delta(s, v)$.

На каждом шаге индукции рассмотрим белую вершину v, которая открывается в процессе поиска из вершины u. Согласно гипотезе индукции $u.d \geq \delta(s, u)$. На основании присвоения в строке 15 и леммы 22.1 получаем

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

После этого вершина v вносится в очередь. Поскольку она при этом окрашивается в серый цвет, а строки 14–17 выполняются только для белых вершин, рассмотренная вершина v больше помещаться в очередь не будет. Таким образом, не будет изменяться и ее значение $v.d$, так что гипотеза индукции выполняется. ∎

Для доказательства того, что $v.d = \delta(s, v)$, сначала разберемся, как работает очередь Q в процедуре BFS. Следующая лемма показывает, что в любой момент времени в очереди находится не более двух различных значений d.

Лемма 22.3

Предположим, что во время выполнения процедуры BFS над графом $G = (V, E)$ очередь Q содержит вершины $\langle v_1, v_2, \ldots, v_r \rangle$, где v_1 — голова Q, а v_r — ее хвост. Тогда для всех $i = 1, 2, \ldots, r - 1$ выполняется $v_r.d \leq v_1.d + 1$ и $v_i.d \leq v_{i+1}.d$.

Доказательство. В доказательстве используется индукция по числу операций с очередью. Изначально, когда в очереди содержится только одна вершина s, лемма, определенно, выполняется.

На каждом шаге индукции необходимо доказать, что лемма выполняется как после помещения вершины в очередь, так и после извлечения ее оттуда. Если из очереди извлекается ее голова v_1, новой головой очереди становится вершина v_2 (если очередь после извлечения некоторой вершины становится пустой, то лемма выполняется исходя из того, что очередь пуста). В соответствии с гипотезой

индукции $v_1.d \le v_2.d$. Но тогда мы имеем $v_r.d \le v_1.d + 1 \le v_2.d + 1$, а все остальные неравенства не изменяются. Следовательно, лемма справедлива, когда новой головой очереди становится v_2.

Чтобы понять, что происходит при внесении вершины в очередь, нужно рассмотреть код более подробно. Когда мы вносим в очередь вершину v в строке 17 процедуры BFS, она становится вершиной v_{r+1}. В этот момент времени вершина u, список смежности которой сканируется, уже удалена из очереди Q, так что согласно гипотезе индукции для новой головы v_1 выполняется неравенство $v_1.d \ge u.d$. Таким образом, $v_{r+1}.d = v.d = u.d + 1 \le v_1.d + 1$. Кроме того, согласно гипотезе индукции $v_r.d \le u.d+1$, так что $v_r.d \le u.d+1 = v.d = v_{r+1}.d$, а остальные неравенства остаются неизменными. Следовательно, лемма выполняется при внесении в очередь новой вершины v. ■

Приведенное ниже следствие показывает, что значения d вносимых в очередь вершин монотонно возрастают со временем.

Следствие 22.4

Предположим, что вершины v_i и v_j вносятся в очередь в процессе работы процедуры BFS и что v_i вносится в очередь до v_j. Тогда в момент внесения в очередь вершины v_j выполняется неравенство $v_i.d \le v_j.d$.

Доказательство. Доказательство вытекает непосредственно из леммы 22.3 и того факта, что каждая вершина получает конечное значение d в процессе выполнения процедуры BFS не более одного раза. ■

Теперь мы можем доказать, что метод поиска в ширину корректно определяет длины кратчайших путей.

Теорема 22.5 (Корректность поиска в ширину)

Пусть $G = (V, E)$ представляет собой ориентированный или неориентированный граф, и предположим, что процедура BFS выполняется над графом G с определенной исходной вершиной $s \in V$. Тогда в процессе работы процедура BFS открывает все вершины $v \in V$, достижимые из исходной вершины s, а по окончании работы для всех $v \in V$ справедливо равенство $v.d = \delta(s, v)$. Кроме того, для всех достижимых из s вершин $v \ne s$ один из кратчайших путей от s к v — это путь от s к $v.\pi$, за которым следует ребро $(v.\pi, v)$.

Доказательство. Предположим, с целью использовать доказательство от обратного, что у некоторой вершины значение d не равно длине кратчайшего пути. Пусть v — вершина с минимальной длиной пути $\delta(s, v)$ среди вершин, для которых оказывается неверным вычисленное значение d. Очевидно, что $v \ne s$. Согласно лемме 22.2 $v.d \ge \delta(s, v)$, так что в силу нашего исходного предположения $v.d > \delta(s, v)$. Вершина v должна быть достижима из s, потому что, если это не так, $\delta(s, v) = \infty \ge v.d$. Пусть u — вершина, непосредственно предшествующая v на кратчайшем пути от s к v, так что $\delta(s, v) = \delta(s, u) + 1$. Поскольку $\delta(s, u) < \delta(s, v)$, в силу нашего выбора вершины v выполняется условие

$u.d = \delta(s,u)$. Объединив полученные результаты, имеем

$$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1 . \qquad (22.1)$$

Теперь рассмотрим момент, когда процедура BFS удаляет узел u из очереди Q в строке 11. В этот момент вершина v может быть белой, серой или черной. Мы покажем, что для каждого из этих случаев мы получим противоречие с неравенством (22.1). Если вершина v белая, то в строке 15 выполняется присвоение $v.d = u.d + 1$, противоречащее (22.1). Если v — черная вершина, то она уже удалена из очереди и согласно следствию 22.4 $v.d \leq u.d$, что, опять-таки, противоречит (22.1). Если v — серая вершина, то она была окрашена в этот цвет при удалении из очереди некоторой вершины w, которая была удалена ранее вершины u и для которой выполняется равенство $v.d = w.d + 1$. Однако из следствия 22.4 вытекает, что $w.d \leq u.d$, поэтому $v.d = w.d + 1 \leq u.d + 1$, что также противоречит (22.1).

Итак, мы заключили, что $v.d = \delta(s,v)$ для всех $v \in V$. Процедурой должны быть открыты все достижимые из s вершины v, потому что, если это не так, для них будет выполняться $\infty = v.d > \delta(s,v)$. Для завершения доказательства теоремы заметим, что если $v.\pi = u$, то $v.d = u.d + 1$. Следовательно, мы можем получить кратчайший путь из s в v, взяв кратчайший путь из s в $v.\pi$, а затем пройдя по ребру $(v.\pi, v)$. ■

Деревья поиска в ширину

Процедура BFS строит в процессе обхода графа дерево поиска в ширину, как показано на рис. 22.3. Это дерево соответствует атрибутам π в каждой вершине. Говоря более формально, для графа $G = (V, E)$ с исходной вершиной s мы определяем ***подграф предшествования*** (predecessor subgraph) графа G как $G_\pi = (V_\pi, E_\pi)$, где

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

и

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} .$$

Подграф предшествования G_π является ***деревом поиска в ширину*** (breadth-first tree), если V_π состоит из вершин, достижимых из s, и если для всех $v \in V_\pi$ в G_π имеется единственный простой путь из s в v, такой что он одновременно является кратчайшим путем из s в v в G. Дерево поиска в ширину является деревом, поскольку оно является связным и $|E_\pi| = |V_\pi| - 1$ (см. теорему Б.2). Ребра в E_π называются ***ребрами дерева*** (tree edges).

Следующая лемма показывает, что после выполнения процедуры BFS над графом G с исходной вершиной s подграф предшествования представляет собой дерево поиска в ширину.

Лемма 22.6

Будучи примененной к ориентированному или неориентированному графу $G = (V, E)$, процедура BFS строит π таким образом, что подграф предшествования $G_\pi = (V_\pi, E_\pi)$ является деревом поиска в ширину.

Доказательство. В строке 16 процедуры BFS присвоение $v.\pi = u$ выполняется тогда и только тогда, когда $(u, v) \in E$ и $\delta(s, v) < \infty$, т.е. если v достижимо из s. Следовательно, V_π состоит из вершин V, достижимых из s. Поскольку G_π образует дерево, согласно теореме Б.2 он содержит единственный путь из s в каждую вершину множества V_π. Индуктивно применив теорему 22.5, мы заключаем, что каждый такой путь является кратчайшим в G. ∎

Приведенная далее процедура выводит все вершины на кратчайшем пути из s в v в предположении, что дерево поиска в ширину уже построено процедурой BFS.

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "Путь из" s "в" v "отсутствует"
else PRINT-PATH(G, s, v.π)
    print v
```

Время работы процедуры линейно зависит от количества выводимых вершин, поскольку каждый рекурсивный вызов процедуры осуществляется для пути, который на одну вершину короче текущего.

Упражнения

22.2.1

Покажите, какие значения d и π получатся в результате поиска в ширину в ориентированном графе, показанном на рис. 22.2, (а), если в качестве исходной взять вершину 3.

22.2.2

Покажите, какие значения d и π получатся в результате поиска в ширину в неориентированном графе, показанном на рис. 22.3, если в качестве исходной взять вершину u.

22.2.3

Покажите, что достаточно одного бита для цвета для того, чтобы процедура BFS давала корректный результат, доказав, что при удалении строки 18 из процедуры она будет давать тот же результат, что и с этой строкой.

22.2.4

Чему будет равно время работы процедуры BFS, адаптированной для работы с матричным представлением графа?

22.2.5

Докажите, что при выполнении поиска в ширину значение $u.d$, назначаемое вершине u, не зависит от порядка перечисления вершин в списках смежности. Используя рис. 22.3 в качестве примера, покажите, что вид дерева поиска в ширину, вычисленного с помощью процедуры BFS, может зависеть от порядка перечисления вершин в списках смежности.

22.2.6

Приведите пример ориентированного графа $G = (V, E)$, исходной вершины $s \in V$ и множества ребер дерева $E_\pi \subseteq E$, таких, что для каждой вершины $v \in V$ единственный путь в графе (V, E_π) от s к v является кратчайшим путем в графе G, но множество ребер E_π невозможно получить с помощью процедуры BFS ни при каком порядке вершин в списках смежности.

22.2.7

Предположим, что есть n борцов, между любой парой которых может состояться (а может и не состояться) поединок, и список r поединков. Разработайте алгоритм, который за время $O(n + r)$ определяет, можно ли разделить всех борцов на две команды так, чтобы в поединках встречались только борцы из разных команд и чтобы, если это возможно, алгоритм выполнял такое разделение.

22.2.8 ★

Диаметр дерева $T = (V, E)$ определяется как $\max_{u,v \in V} \delta(u, v)$, т.е. диаметр — это наибольшая длина кратчайшего пути в дереве. Разработайте эффективный алгоритм вычисления диаметра дерева и проанализируйте время его работы.

22.2.9

Пусть $G = (V, E)$ — связный неориентированный граф. Разработайте алгоритм для вычисления за время $O(V + E)$ пути в G, который проходит по каждому ребру из E ровно по одному разу в каждом направлении. Придумайте способ выйти из лабиринта, если у вас в карманах имеется много монет.

22.3. Поиск в глубину

Стратегия поиска в глубину, как следует из ее названия, состоит в том, чтобы идти "в глубь" графа, насколько это возможно. При выполнении поиска в глубину он исследует все ребра, выходящие из вершины, открытой последней, и покидает вершину, только когда не остается неисследованных ребер — при этом происходит "откат" в вершину, из которой была открыта вершина v. Этот процесс продолжается до тех пор, пока не будут открыты все вершины, достижимые из исходной.

Если при этом остаются неоткрытые вершины, то одна из них выбирается в качестве новой исходной вершины и поиск повторяется уже из нее. Этот процесс повторяется до тех пор, пока не будут открыты все вершины.[3]

Как и в случае поиска в ширину, когда вершина v открывается в процессе сканирования списка смежности уже открытой вершины u, процедура поиска записывает это событие, устанавливая атрибут $v.\pi$ равным u. В отличие от поиска в ширину, когда подграф предшествования образует дерево, при поиске в глубину подграф предшествования может состоять из нескольких деревьев, так как поиск может выполняться из нескольких исходных вершин. ***Подграф предшествования*** (predecessor subgraph) поиска в глубину, таким образом, несколько отличается от такового для поиска в ширину. Мы определяем его как $G_\pi = (V, E_\pi)$, где

$$E_\pi = \{(v.\pi, v) : v \in V \text{ и } v.\pi \neq \text{NIL}\} \ .$$

Подграф предшествования поиска в глубину образует ***лес поиска в глубину*** (depth-first forest), который состоит из нескольких ***деревьев поиска в глубину*** (depth-first trees). Ребра в E_π называются ***ребрами дерева*** (tree edges).

Как и в процессе поиска в ширину, вершины графа окрашиваются в разные цвета, свидетельствующие об их состоянии. Каждая вершина изначально белая, затем при ***открытии*** (discover) в процессе поиска она окрашивается в серый цвет, и по ***завершении***, когда ее список смежности полностью просканирован, она становится черной. Такая методика гарантирует, что каждая вершина в конечном счете находится только в одном дереве поиска в глубину, так что деревья не пересекаются.

Помимо построения леса поиска в глубину, поиск в глубину также проставляет в вершинах ***метки времени*** (timestamp). Каждая вершина имеет две такие метки: первую — $v.d$, в которой записывается, когда вершина v открывается (и окрашивается в серый цвет), и вторую — $v.f$, которая фиксирует момент, когда поиск завершает сканирование списка смежности вершины v и она становится черной. Эти метки используются многими алгоритмами и полезны при рассмотрении поведения поиска в глубину.

Приведенная ниже процедура DFS записывает в атрибут $u.d$ момент, когда вершина u открывается, а в атрибут $u.f$ — момент завершения работы с вершиной u. Эти метки времени представляют собой целые числа в диапазоне от 1 до $2|V|$, поскольку для каждой из $|V|$ вершин имеется только одно событие открытия и одно — завершения. Для каждой вершины u

$$u.d < u.f \ . \tag{22.2}$$

[3]Может показаться несколько странным то, что поиск в ширину ограничен только одной исходной вершиной, в то время как поиск в глубину может выполняться из иескольких исходных вершин. Хотя концептуально поиск в ширину может выполняться из нескольких исходных вершин, а поиск в глубину — быть ограниченным одной исходной вершиной, такой подход отражает типичное использование результатов поиска. Поиск в ширину обычно используется для определения длин кратчайших путей (и связанного с ними графа предшествования) из данной вершины. Поиск в глубину, как мы увидим позже в этой главе, чаще используется в качестве подпрограммы в других алгоритмах.

До момента времени $u.d$ вершина u имеет цвет WHITE, между $u.d$ и $u.f$ — цвет GRAY, а после — цвет BLACK.

Далее представлен псевдокод алгоритма поиска в глубину. Входной граф G может быть как ориентированным, так и неориентированным. Переменная *time* — глобальная и используется нами для меток времени.

```
DFS(G)
1  for каждой вершины u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for каждой вершины u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1              // Открыта белая вершина u
 2  u.d = time
 3  u.color = GRAY
 4  for каждой v ∈ G.Adj[u]      // Исследование ребра (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // Завершение работы с u
 9  time = time + 1
10  u.f = time
```

На рис. 22.4 показана работа процедуры DFS с графом на рис. 22.2.

Процедура DFS работает следующим образом. В строках 1–3 все вершины окрашиваются в белый цвет, а их поля π инициализируются значением NIL. В строке 4 выполняется сброс глобального счетчика времени. В строках 5–7 поочередно проверяются все вершины из V, и когда обнаруживается белая вершина, она обрабатывается с помощью процедуры DFS-VISIT. Каждый раз при вызове процедуры DFS-VISIT(G, u) в строке 7 вершина u становится корнем нового дерева леса поиска в глубину. При возврате из процедуры DFS каждой вершине u сопоставляются два момента времени — ***время открытия*** (discovery time) $u.d$ и ***время завершения*** (finishing time) $u.f$.

При каждом вызове DFS-VISIT(G, u) вершина u изначально имеет белый цвет. В строке 1 увеличивается глобальная переменная *time*, в строке 2 выполняется запись нового значения переменной *time* в атрибут времени открытия $u.d$, а в строке 3 вершина u окрашивается в серый цвет. В строках 4–7 исследуются все вершины v, смежные с u, и выполняется рекурсивное посещение всех вершин v, являющихся белыми. При рассмотрении в строке 4 каждой вершины $v \in Adj[u]$ мы говорим, что ребро (u, v) ***исследуется*** (explored) поиском в глубину. И наконец, после того как будут исследованы все ребра, покидающие u,

21 Зак. 3726

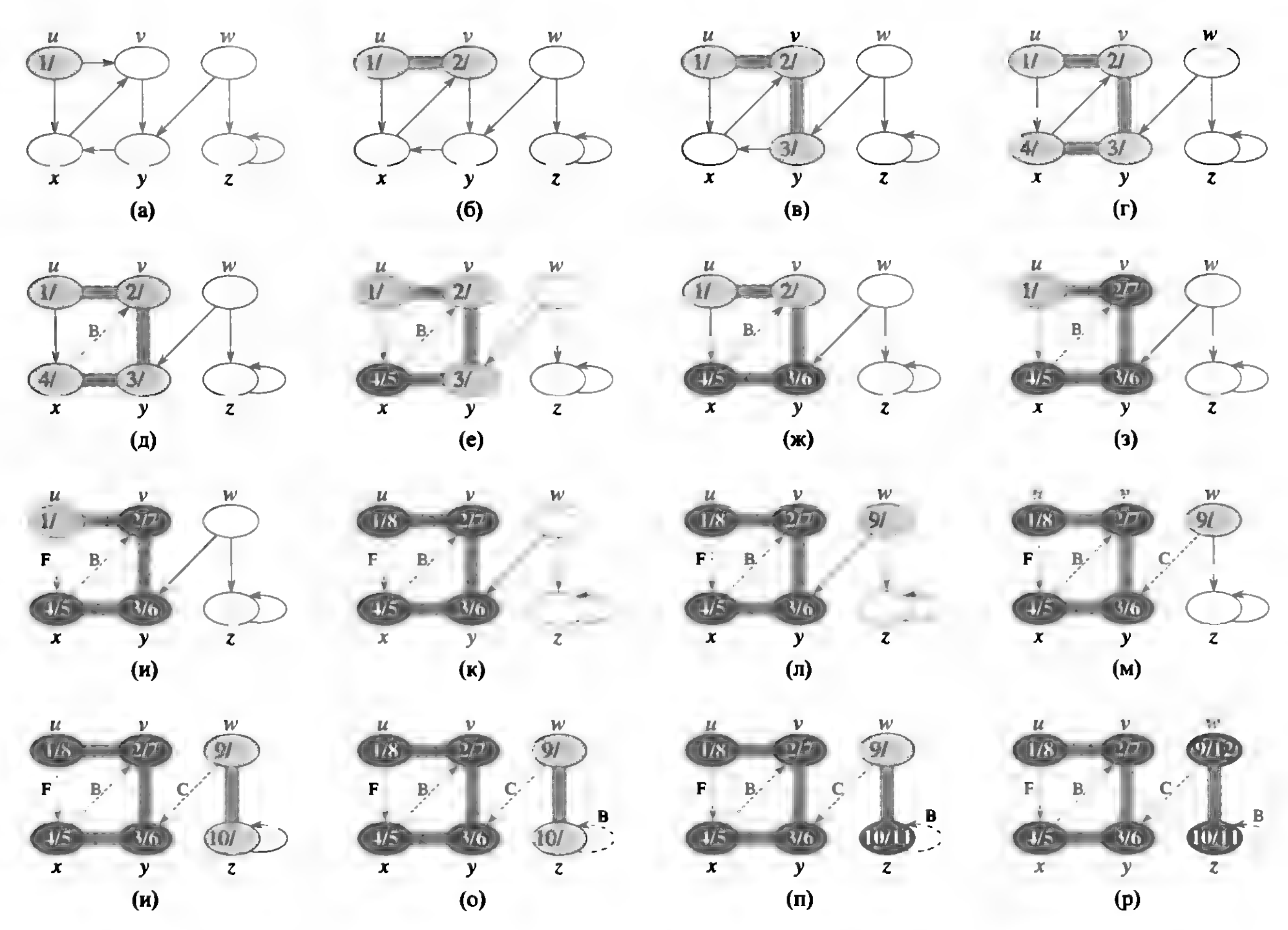


Рис. 22.4. Применение алгоритма поиска в глубину DFS к ориентированному графу. Исследуемые при этом ребра показаны затененными (если это ребра дерева) или пунктирными (в противном случае). Ребра, не являющиеся ребрами дерева, помечены буквами B, C или F в соответствии с тем, являются ли они обратными (back), перекрестными (cross) или прямыми (forward). В вершинах указаны метки времени в формате "открытие/завершение".

в строках 8–10 вершина u окрашивается в черный цвет, а в атрибут $u.f$ записывается время завершения работы с ней.

Заметим, что результат поиска в глубину может зависеть от порядка, в котором выполняется рассмотрение вершин в строке 5 процедуры DFS, а также от порядка посещения смежных вершин в строке 4 процедуры DFS-VISIT. На практике это обычно не вызывает каких-либо проблем, так как обычно эффективно использоваться может *любой* результат поиска в глубину, приводя, по сути, к одинаковым результатам работы алгоритма, опирающегося на поиск в глубину.

Чему равно время работы процедуры DFS? Циклы в строках 1–3 и 5–7 процедуры DFS выполняются за время $\Theta(V)$, исключая время, необходимое для вызова процедуры DFS-VISIT. Как и в случае поиска в ширину, воспользуемся групповым анализом. Процедура DFS-VISIT вызывается ровно по одному разу для каждой вершины $v \in V$, так как она вызывается только для белых вершин, а первое, что она делает, — это окрашивает переданную в качестве параметра вершину в серый цвет. В процессе выполнения DFS-VISIT(G, v) цикл в строках 4–7 вы-

полняется $|Adj[v]|$ раз. Поскольку

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

общая стоимость выполнения строк 4–7 процедуры DFS-VISIT равна $\Theta(E)$. Время работы процедуры DFS, таким образом, равно $\Theta(V + E)$.

Свойства поиска в глубину

Поиск в глубину дает нам важную информацию о структуре графа. Вероятно, наиболее фундаментальное свойство поиска в глубину заключается в том, что подграф предшествования G_π в действительности образует лес деревьев, поскольку структура деревьев поиска в глубину в точности отражает структуру рекурсивных вызовов процедуры DFS-VISIT. То есть $u = v.\pi$ тогда и только тогда, когда DFS-VISIT(G, v) была вызвана при сканировании списка смежности вершины u. Кроме того, вершина v является потомком вершины u в лесу поиска в глубину тогда и только тогда, когда вершина u была серой в момент открытия вершины v.

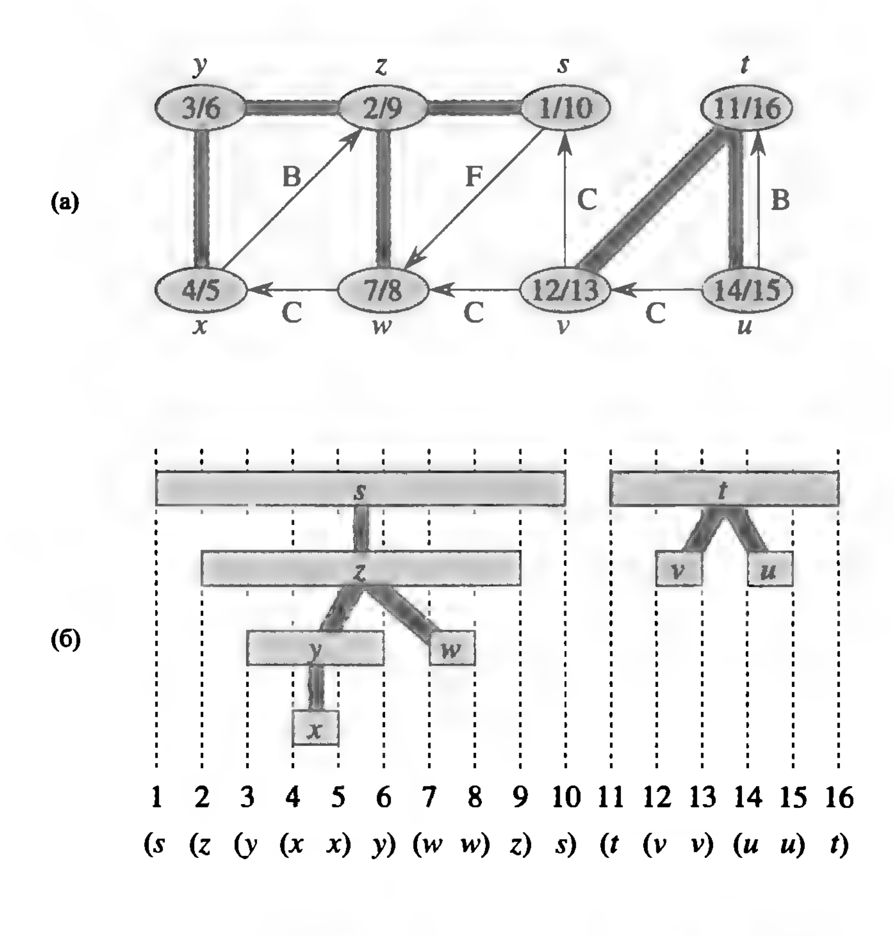
Еще одно важное свойство поиска в глубину заключается в том, что времена открытия и завершения образуют ***скобочную структуру*** (parenthesis structure). Если мы представим открытие вершины u с помощью отрывающейся скобки "(u", а завершение — с помощью закрывающейся скобки "u)", то перечень открытий и завершений образует корректное выражение в смысле вложенности скобок. Например, поиск в глубину на рис. 22.5, (а) соответствует скобочному выражению на рис. 22.5, (б). Еще одно утверждение о скобочной структуре приведено в следующей теореме.

Теорема 22.7 (Теорема о скобках)

В любом поиске в глубину в (ориентированном или неориентированном) графе $G = (V, E)$ для любых двух вершин u и v выполняется ровно одно из трех следующих утверждений.

- Отрезки $[u.d, u.f]$ и $[v.d, v.f]$ не пересекаются, и ни u не является потомком v в лесу поиска в глубину, ни v не является потомком u.
- Отрезок $[u.d, u.f]$ полностью содержится в отрезке $[v.d, v.f]$, а u является потомком v в дереве поиска в глубину.
- Отрезок $[v.d, v.f]$ полностью содержится в отрезке $[u.d, u.f]$, а v является потомком u в дереве поиска в глубину.

Доказательство. Начнем со случая, когда $u.d < v.d$. При этом необходимо рассмотреть два подслучая в зависимости от того, справедливо ли неравенство $v.d < u.f$. Первый подслучай соответствует справедливости неравенства $v.d < u.f$, так что вершина v была открыта, когда вершина u все еще была окрашена в серый цвет. Отсюда следует, что v является потомком u. Кроме того, поскольку вершина v открыта позже, чем u, перед тем как вернуться для завер-



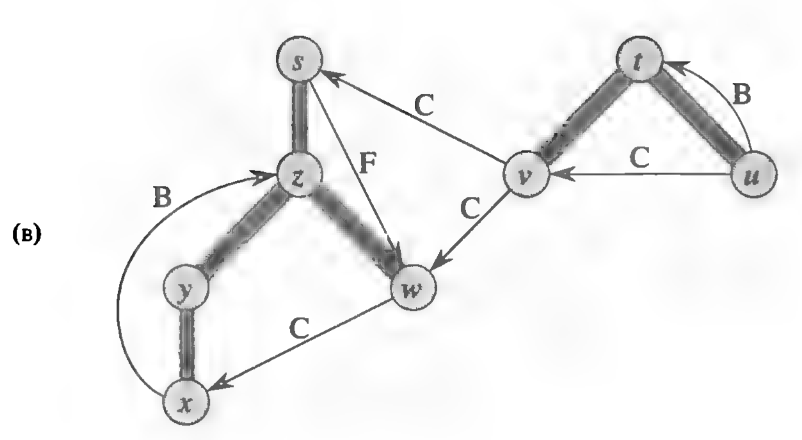


Рис. 22.5. Свойства поиска в глубину. **(а)** Результат поиска в глубину в ориентированном графе. Метки времени вершин и типы ребер указаны, как на рис. 22.4. **(б)** Интервалы между временами открытия и завершения для каждой вершины соответствуют показанной структуре скобок. Каждый прямоугольник охватывает интервал, задаваемый временами открытия и завершения соответствующей вершины. Показаны только три ребра. Если два интервала перекрываются, то один из них вложен в другой, и вершина, соответствующая меньшему интервалу, является потомком вершины, соответствующей большему интервалу. **(в)** Граф из части (а), перерисованный таким образом, чтобы ребра деревьев и прямые ребра были направлены вниз, а обратные шли вверх от потомков к предкам.

шения поиска к вершине u, алгоритмом будут исследованы все выходящие из v ребра. В таком случае, следовательно, отрезок $[v.d, v.f]$ полностью содержится в отрезке $[u.d, u.f]$. В другом подслучае $u.f < v.d$, и из неравенства (22.2) следует, что $u.d < u.f < v.d < v.f$; таким образом, отрезки $[u.d, u.f]$ и $[v.d, v.f]$ не пересекаются. Поскольку отрезки не пересекаются, открытие одной вершины не происходит до тех пор, пока другая имеет серый цвет, так что ни одна вершина не является потомком другой.

Случай, когда $v.d < u.d$, рассматривается аналогично, происходит только смена ролей u и v. ■

Следствие 22.8 (Вложенность интервалов потомков)

Вершина v является истинным (отличным от самого u) потомком u в лесу поиска в глубину (ориентированного или неориентированного) графа G тогда и только тогда, когда $u.d < v.d < v.f < u.f$.

Доказательство. Непосредственно вытекает из теоремы 22.7. ■

Следующая теорема дает еще одно важное указание о том, когда одна вершина в лесу поиска в глубину является потомком другой.

Теорема 22.9 (Теорема о белом пути)

В лесу поиска в глубину (ориентированного или неориентированного) графа $G = (V, E)$ вершина v является потомком вершины u тогда и только тогда, когда в момент времени $u.d$ (открытие вершины u) вершина v достижима из u по пути, состоящему только из белых вершин.

Доказательство. $\Rightarrow$: Если $v = u$, то путь от u к v содержит единственную вершину u, которая все еще является белой в момент, когда мы устанавливаем значение атрибута $u.d$. Теперь предположим, что v является истинным потомком u в лесу поиска в глубину. Согласно следствию 22.8 $u.d < v.d$, и, таким образом, вершина v является белой в момент $u.d$. Поскольку v может быть любым потомком u, все вершины на единственном простом пути от u до v в лесу поиска в глубину в момент $u.d$ белые.

$\Leftarrow$: Предположим, что в момент времени $u.d$ вершина v достижима из u вдоль пути, состоящего из белых вершин, но v не становится потомком u в дереве поиска в глубину. Без потери общности предположим, что все остальные вершины вдоль пути становятся потомками u (в противном случае в качестве v выберем на пути ближайшую к u вершину, которая не становится потомком u). Пусть w — предшественник v на пути, так что w является потомком u (w и u в действительности могут быть одной и той же вершиной). Согласно следствию 22.8 $w.f \leq u.f$. Поскольку вершина v должна быть открыта после того, как открыта u, но перед тем, как завершена обработка w, мы имеем $u.d < v.d < w.f \leq u.f$. Тогда из теоремы 22.7 следует, что отрезок $[v.d, v.f]$ полностью содержится внутри отрезка $[u.d, u.f]$. Согласно следствию 22.8 вершина v должна в конечном итоге быть потомком вершины u. ■

Классификация ребер

Еще одно интересное свойство поиска в глубину заключается в том, что поиск может использоваться для классификации ребер входного графа $G = (V, E)$. Эта классификация ребер может использоваться для получения важной информации о графе. Например, в следующем разделе мы увидим, что ориентированный граф ацикличен тогда и только тогда, когда при поиске в глубину в нем не обнаруживается "обратных" ребер (лемма 22.11).

Мы можем определить четыре типа ребер в терминах леса G_π, полученного при поиске в глубину в графе G.

1. ***Ребра деревьев*** (tree edges) — это ребра леса G_π. Ребро (u, v) является ребром дерева, если при исследовании этого ребра была впервые открыта вершина v.
2. ***Обратные ребра*** (back edges) — это ребра (u, v), соединяющие вершину u с ее предком v в дереве поиска в глубину. Петли (ребра-циклы), которые могут встречаться в ориентированных графах, рассматриваются как обратные ребра.
3. ***Прямые ребра*** (forward edges) — это ребра (u, v), не являющиеся ребрами дерева и соединяющие вершину u с ее потомком v в дереве поиска в глубину.
4. ***Перекрестные ребра*** (cross edges) — все остальные ребра графа. Они могут соединять вершины одного и того же дерева поиска в глубину, когда ни одна из вершин не является предком другой, или соединять вершины в разных деревьях поиска в глубину.

На рис. 22.4 и 22.5 ребра помечены в соответствии с их типом. На рис. 22.5, (в) показан граф с рис. 22.5, (а), перерисованный так, что все прямые ребра и ребра деревьев направлены вниз, а обратные ребра — вверх. Таким способом можно перерисовать любой граф.

Алгоритм DFS можно модифицировать так, что он будет классифицировать встречающиеся при работе ребра. Ключевая идея состоит в том, что каждое ребро (u, v) можно классифицировать с помощью цвета вершины v при первом его исследовании (правда, при этом не различаются прямые и перекрестные ребра).

1. WHITE говорит о том, что это ребро дерева.
2. GRAY определяет обратное ребро.
3. BLACK указывает на прямое или перекрестное ребро.

Первый случай следует непосредственно из определения алгоритма. Рассматривая второй случай, заметим, что серые вершины всегда образуют линейную цепочку потомков, соответствующую стеку активных вызовов процедуры DFS-VISIT; количество серых вершин на единицу больше глубины последней открытой вершины в дереве поиска в глубину. Исследование всегда начинается с самой глубокой серой вершины, так что ребро, которое достигает другой серой вершины, достигает предка исходной вершины. В третьем случае мы имеем дело с остальными ребрами, не подпадающими под первый или второй случай. Можно показать, что ребро (u, v) является прямым, если $u.d < v.d$, и перекрестным, если $u.d > v.d$ (см. упр. 22.3.5).

В неориентированном графе при классификации ребер может возникнуть определенная неоднозначность, поскольку (u, v) и (v, u) в действительности являются одним ребром. В таком случае, когда ребро соответствует нескольким категориям, оно классифицируется в соответствии с *первой* категорией в списке, применимой для данного ребра. Тот же результат получается, если выполнять классификацию в соответствии с тем, в каком именно виде — (u, v) или (v, u) — ребро встречается в первый раз в процессе выполнения алгоритма (см. упр. 22.3.6).

Теперь мы покажем, что прямые и перекрестные ребра никогда не встречаются при поиске в глубину в неориентированном графе.

Теорема 22.10

При поиске в глубину в неориентированном графе G любое ребро G является либо ребром дерева, либо обратным ребром.

Доказательство. Пусть (u, v) — произвольное ребро графа G, и предположим без потери общности, что $u.d < v.d$. Тогда вершина v должна быть открыта и завершена до того, как будет завершена работа с вершиной u (пока u — серая), так как v находится в списке смежности u. Если ребро (u, v) исследуется сначала в направлении от u к v, то v до этого момента была неоткрытой (белой) — так как в противном случае мы бы уже исследовали это ребро в направлении от v к u. Таким образом, (u, v) становится ребром дерева. Если же ребро (u, v) исследуется сначала в направлении от v к u, то оно является обратным, поскольку вершина u при первом исследовании ребра — серая. ■

Мы ознакомимся с некоторыми применениями этой теоремы в последующих разделах.

Упражнения

22.3.1

Нарисуйте таблицу размером 3×3 со строками и столбцами, помеченными как WHITE (белый), GRAY (серый) и BLACK (черный). В каждой ячейке (i, j) пометьте, может ли быть обнаружено в процессе поиска в глубину в ориентированном графе ребро из вершины цвета i в вершину цвета j. Для каждого возможного ребра укажите его возможный тип. Постройте такую же таблицу для неориентированного графа.

22.3.2

Покажите, как работает поиск в глубину для графа, изображенного на рис. 22.6. Считаем, что цикл **for** в строках 5–7 процедуры DFS сканирует вершины в алфавитном порядке, а также что все списки смежности упорядочены по алфавиту. Для каждой вершины укажите время открытия и завершения, а для каждого ребра — его тип.

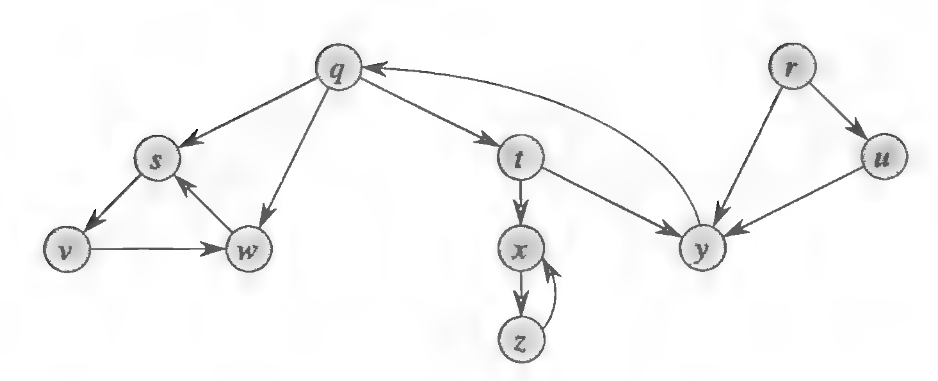


Рис. 22.6. Ориентированный граф к упр. 22.3.2 и 22.5.2.

22.3.3

Напишите скобочное выражение, соответствующее поиску в глубину, показанному на рис. 22.4.

22.3.4

Покажите, что в каждой вершине достаточно только одного бита для хранения цвета, доказав, что результат работы процедуры DFS не изменится при удалении строки 8 из процедуры DFS-VISIT.

22.3.5

Покажите, что ребро (u, v) является

а. ребром дерева или прямым ребром тогда и только тогда, когда $u.d < v.d < v.f < u.f$;

б. обратным ребром тогда и только тогда, когда $v.d \le u.d < u.f \le v.f$;

в. перекрестным ребром тогда и только тогда, когда $v.d < v.f < u.d < u.f$.

22.3.6

Покажите, что в неориентированном графе классификация ребра (u, v) как ребра дерева или обратного ребра в зависимости от того, встречается ли первым ребро (u, v) или (v, u) при поиске в глубину, эквивалентна классификации в соответствии с приоритетами типов в схеме классификации.

22.3.7

Перепишите процедуру DFS, используя стек для устранения рекурсии.

22.3.8

Приведите контрпример к гипотезе о том, что если в ориентированном графе G имеется путь от u к v и что если $u.d < v.d$ при поиске в глубину в G, то v — потомок u в полученном лесу поиска в глубину.

22.3.9

Приведите контрпример к гипотезе, заключающейся в том, что если в ориентированном графе G имеется путь от u к v, то любой поиск в глубину должен дать в результате $v.d \le u.f$.

22.3.10

Модифицируйте псевдокод поиска в глубину так, чтобы он выводил все ребра ориентированного графа G вместе с их типами. Какие изменения следует внести в псевдокод (если таковые требуются) для работы с неориентированным графом?

22.3.11

Объясните, как вершина u ориентированного графа может оказаться единственной в дереве поиска в глубину, несмотря на наличие у нее как входящих, так и исходящих ребер в G.

22.3.12

Покажите, что поиск в глубину в неориентированном графе G может использоваться для определения связных компонентов графа G и что лес поиска в глубину содержит столько деревьев, сколько в графе связных компонентов. Говоря более точно, покажите, как изменить поиск в глубину так, чтобы каждой вершине v присваивалась целочисленная метка $v.cc$ в диапазоне от 1 до k (где k — количество связных компонентов в G), такая, что $u.cc = v.cc$ тогда и только тогда, когда u и v принадлежат одному и тому же связному компоненту.

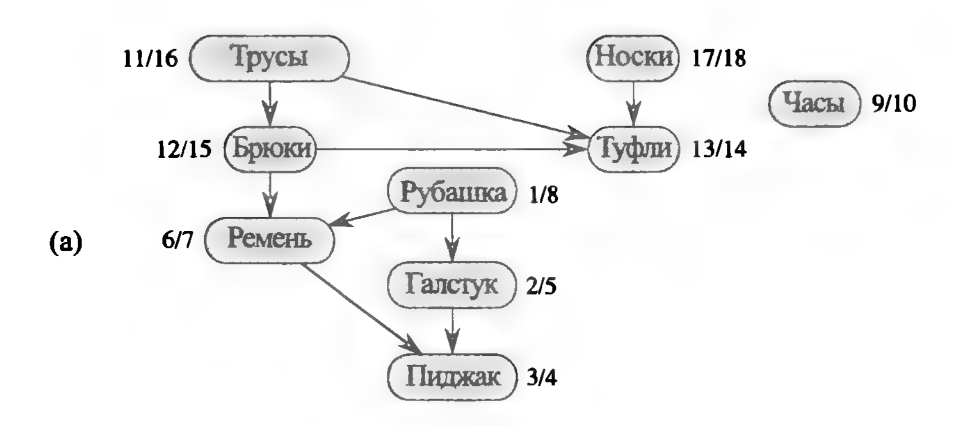
22.3.13 ★

Ориентированный граф $G = (V, E)$ обладает свойством ***односвязности*** (singly connected), если из $u \leadsto v$ следует, что в G имеется не более одного пути от u к v для всех вершин $u, v \in V$. Разработайте эффективный алгоритм для определения, является ли ориентированный граф односвязным.

22.4. Топологическая сортировка

В этом разделе показано, каким образом можно использовать поиск в глубину для топологической сортировки ориентированного ациклического графа (directed acyclic graph, для которого иногда используется аббревиатура "dag"). ***Топологическая сортировка*** (topological sort) ориентированного ациклического графа $G = (V, E)$ представляет собой такое линейное упорядочение всех его вершин, что если граф G содержит ребро (u, v), то u при таком упорядочении располагается до v (если граф содержит цикл, такая сортировка невозможна). Топологическую сортировку графа можно рассматривать как такое упорядочение его вершин вдоль горизонтальной линии, что все ребра направлены слева направо. Таким образом, топологическая сортировка существенно отличается от обычных видов сортировки, рассмотренных в части II.

Ориентированные ациклические графы используются во многих приложениях для указания последовательности событий. На рис. 22.7 приведен пример графа, построенного профессором Рассеянным для утреннего одевания. Одни вещи обязательно нужно одевать раньше других, например сначала носки, а затем туфли. Другие вещи могут быть одеты в произвольном порядке (например, носки и рубашка). Ребро (u, v) в ориентированном ациклическом графе на рис. 22.7, (а) по-



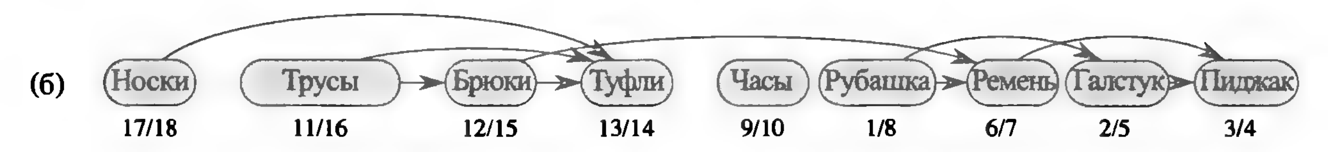


Рис. 22.7. **(а)** Топологическая сортировка процесса одевания профессора Рассеянного. Каждое ориентированное ребро (u, v) означает, что вещь u должна быть одета до вещи v. Для каждой вершины приведены время открытия и время завершения. **(б)** Тот же граф, изображенный с учетом выполненной топологической сортировки; вершины располагаются слева направо в порядке уменьшения времени завершения. Все ориентированные ребра направлены слева направо.

казывает, что вещь u должна быть одета раньше вещи v. Топологическая сортировка этого графа дает нам порядок одевания. На рис. 22.7, (б) показан отсортированный ориентированный ациклический граф, вершины которого расположены вдоль горизонтальной линии так, что все ребра направлены слева направо.

Вот простой алгоритм топологической сортировки ориентированного ациклического графа.

```
TOPOLOGICAL-SORT(G)
1  Вызвать DFS(G) для вычисления времен завершения v.f
          для каждой вершины v
2  По завершении работы над вершиной внести ее в начало
          связанного списка
3  return связанный список вершин
```

На рис. 22.7, (б) видно, что топологически отсортированные вершины располагаются в порядке убывания времени завершения.

Мы можем выполнить топологическую сортировку за время $\Theta(V + E)$, поскольку поиск в глубину выполняется именно за это время, а вставка каждой из $|V|$ вершин в начало связанного списка занимает время $O(1)$.

Докажем корректность этого алгоритма с использованием следующей ключевой леммы, характеризующей ориентированный ациклический граф.

Лемма 22.11

Ориентированный граф G является ациклическим тогда и только тогда, когда поиск в глубину в G не находит в нем обратных ребер.

Доказательство. $\Rightarrow$: Предположим, что имеется обратное ребро (u, v). Тогда вершина v является предком u в лесу поиска в глубину. Таким образом, в графе G имеется путь от v к u, и обратное ребро (u, v) завершает цикл.

$\Leftarrow$: Предположим, что граф G содержит цикл c. Покажем, что поиск в глубину обнаружит обратное ребро. Пусть v — первая вершина, открытая в c, и пусть (u, v) — предыдущее ребро в c. В момент времени $v.d$ вершины c образуют путь из белых вершин от v к u. В соответствии с теоремой о белом пути вершина u становится потомком v в лесу поиска в глубину. Следовательно, (u, v) — обратное ребро. ■

Теорема 22.12

Процедура TOPOLOGICAL-SORT выполняет топологическую сортировку переданного ей ориентированного ациклического графа.

Доказательство. Предположим, что над данным ориентированным ациклическим графом $G = (V, E)$ выполняется процедура DFS, которая вычисляет времена завершения для его вершин. Достаточно показать, что если для произвольной пары разных вершин $u, v \in V$ в графе G имеется ребро от u к v, то $v.f < u.f$. Рассмотрим произвольное ребро (u, v), исследуемое процедурой DFS(G). При исследовании вершина v не может быть серой, поскольку тогда v была бы предком u и ребро (u, v) представляло бы собой обратное ребро, что противоречит лемме 22.11. Следовательно, вершина v должна быть либо белой, либо черной. Если вершина v — белая, то она становится потомком u, так что $v.f < u.f$. Если v — черная, значит, работа с ней уже завершена и значение $v.f$ уже установлено. Поскольку мы все еще работаем с вершиной u, значение $u.f$ еще не определено, так что, когда это будет сделано, будет выполняться неравенство $v.f < u.f$. Таким образом, для любого ребра (u, v) ориентированного ациклического графа выполняется условие $v.f < u.f$, что и доказывает данную теорему. ■

Упражнения

22.4.1

Покажите, в каком порядке расположит вершины представленного на рис. 22.8 ориентированного ациклического графа процедура TOPOLOGICAL-SORT, если считать, что выполняются все предположения, сформулированные в упр. 22.3.2.

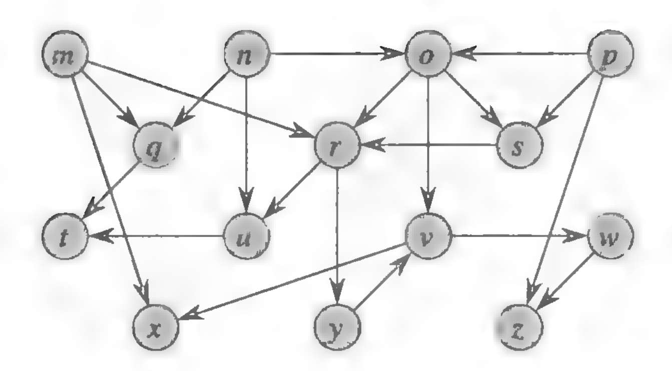


Рис. 22.8. Ориентированный ациклический граф для топологической сортировки.

22.4.2

Разработайте алгоритм с линейным временем работы, который для данного ориентированного ациклического графа $G = (V, E)$ и вершин s и t определяет количество простых путей от s к t в графе G. Например, в графе на рис. 22.8 имеется ровно четыре пути от вершины p к вершине v: pov, $poryv$, $posryv$ и $psryv$. (Ваш алгоритм должен только указать количество путей, не перечисляя их.)

22.4.3

Разработайте алгоритм для определения, содержит ли данный неориентированный граф $G = (V, E)$ простой цикл. Ваш алгоритм должен выполняться за время $O(V)$ независимо от $|E|$.

22.4.4

Докажите или опровергните следующее утверждение: если ориентированный граф G содержит циклы, то процедура TOPOLOGICAL-SORT(G) упорядочивает вершины таким образом, что при этом количество "плохих" ребер (идущих в противоположном направлении) минимально.

22.4.5

Еще один алгоритм топологической сортировки ориентированного ациклического графа $G = (V, E)$ состоит в поиске вершины с входящей степенью 0, ее выводе, удалении из графа этой вершины и всех исходящих из нее ребер и повторении этих действий до тех пор, пока в графе остается хоть одна вершина. Покажите, как реализовать описанный алгоритм, чтобы время его работы составляло $O(V + E)$. Что произойдет, если в графе G будут иметься циклы?

22.5. Сильно связные компоненты

Теперь рассмотрим классическое применение поиска в глубину: разложение ориентированного графа на сильно связные компоненты. В этом разделе будет показано, как выполнить такое разложение с помощью двух поисков в глубину. Ряд алгоритмов для работы с графами начинается с выполнения такого разложения графа, и после разложения алгоритм работает с каждым сильно связным компонентом отдельно. Полученные решения затем комбинируются в соответствии со структурой связей между сильно связными компонентами.

В приложении Б сказано, что сильно связный компонент ориентированного графа $G = (V, E)$ представляет собой максимальное множество вершин $C \subseteq V$, такое, что для каждой пары вершин u и v из C справедливо как $u \leadsto v$, так и $v \leadsto u$, т.е. вершины u и v достижимы одна из другой. На рис. 22.9 приведен соответствующий пример.

Наш алгоритм поиска сильно связных компонентов графа $G = (V, E)$ использует транспонирование G, которое определяется в упр. 22.1.3 как граф $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, где $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$, т.е. E^{T} состоит из ребер G с изменением их направления на обратное. Для представления с использованием списков смеж-

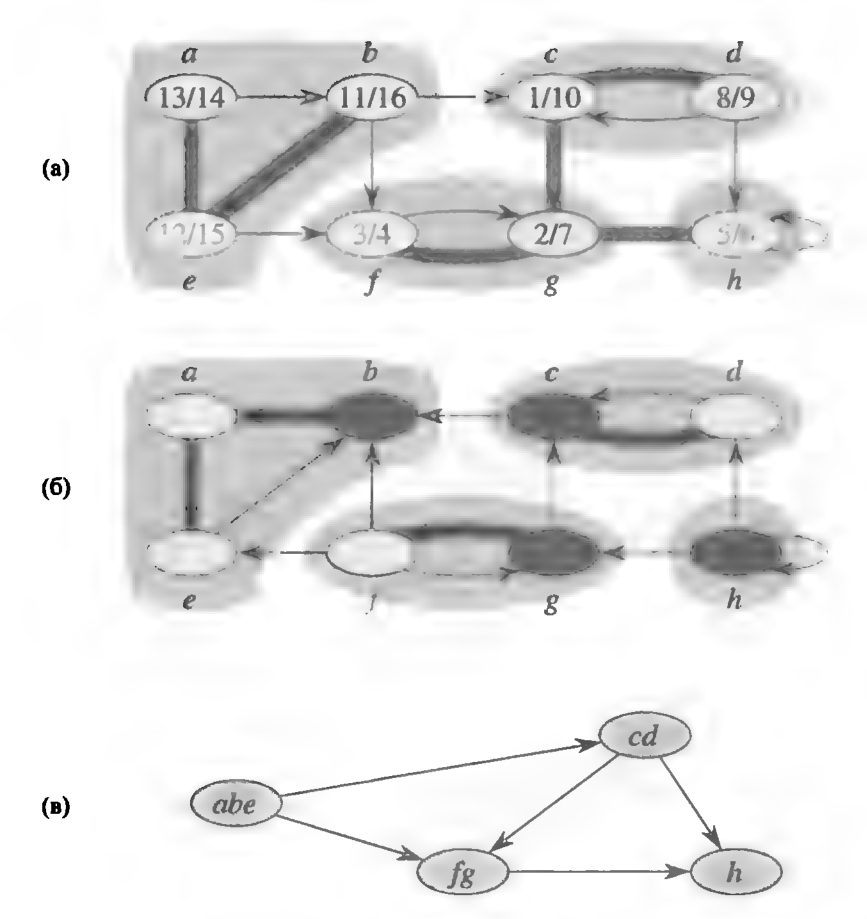


Рис. 22.9. **(а)** Ориентированный граф G. Каждая заштрихованная область представляет собой сильно связный компонент G. Каждая вершина помечена ее временем открытия и завершения при поиске в глубину, а ребра дерева выделены штриховкой. **(б)** Граф G^{T}, транспонированный G, с показанным лесом поиска в глубину, вычисленным в строке 3 процедуры STRONGLY-CONNECTED-COMPONENTS, и выделенными штриховкой ребрами дерева. Каждый сильно связный компонент соответствует одному дереву поиска в глубину. Вершины b, c, g и h, выделенные темным цветом, представляют собой корни деревьев поиска в глубину, сгенерированных поиском в глубину в графе G^{T}. **(в)** Ациклический граф компонентов G^{SCC}, полученный путем сжатия всех ребер в пределах сильно связных компонентов G, так что в каждом компоненте остается только по одной вершине.

ности данного графа G получить граф G^{T} можно за время $O(V + E)$. Интересно заметить, что графы G и G^{T} имеют одни и те же сильно связные компоненты: u и v достижимы одна из другой в G тогда и только тогда, когда они достижимы одна из другой в G^{T}. На рис. 22.9,(б) показан граф, представляющий собой результат транспонирования графа на рис. 22.9,(а) (сильно связные компоненты выделены штриховкой).

Далее приведен алгоритм, который за линейное время $\Theta(V + E)$ находит сильно связные компоненты ориентированного графа $G = (V, E)$ благодаря двойному поиску в глубину: одному — в графе G и второму — в графе G^{T}.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  Вызов DFS(G) для вычисления времен завершения u.f
       для каждой вершины u
2  Вычисление G^T
3  Вызов DFS(G^T), но в основном цикле процедуры DFS
       вершины рассматриваются в порядке убывания значений u.f,
       вычисленных в строке 1
4  Вывод вершин каждого дерева в лесу поиска в глубину,
       полученного в строке 3, в качестве
       отдельного сильно связного компонента
```

Идея, лежащая в основе этого алгоритма, опирается на ключевое свойство ***графа компонентов*** (component graph) $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, который определяется следующим образом. Предположим, что G имеет сильно связные компоненты $C_1, C_2, \ldots, C_k$. Множество вершин $V^{\mathrm{SCC}} = \{v_1, v_2, \ldots, v_k\}$ содержит вершину v_i для каждого сильно связного компонента C_i графа G. Если в G имеется ребро (x, y) для некоторых двух вершин, $x \in C_i$ и $y \in C_j$, то в графе компонентов имеется ребро $(v_i, v_j) \in E^{\mathrm{SCC}}$. Другими словами, если сжать все ребра между смежными вершинами в каждом сильно связном компоненте графа G, мы получим граф G^{SCC} (вершинами которого являются сильно связные компоненты графа G). На рис. 22.9, (в) показан граф компонентов для графа, приведенного на рис. 22.9, (а).

Ключевое свойство графа компонентов состоит в том, что он представляет собой ориентированный ациклический граф, как следует из приведенной ниже леммы.

Лемма 22.13

Пусть C и C' — различные сильно связные компоненты в ориентированном графе $G = (V, E)$ и пусть $u, v \in C$ и $u', v' \in C'$, а кроме того, предположим, что в G имеется путь $u \leadsto u'$. В таком случае в G не может быть пути $v' \leadsto v$.

Доказательство. Если в G имеется путь $v' \leadsto v$, то в G имеются и пути $u \leadsto u' \leadsto v'$ и $v' \leadsto v \leadsto u$. Следовательно, вершины u и v' достижимы одна из другой, что противоречит предположению о том, что C и C' — различные сильно связные компоненты. ■

Мы увидим, что при рассмотрении вершин в процессе второго поиска в глубину в порядке убывания времен завершения работы с вершинами, которые были вычислены при первом поиске в глубину, мы, по сути, посещаем вершины графа компонентов (каждая из которых соответствует сильно связному компоненту G) в порядке топологической сортировки.

Поскольку процедура STRONGLY-CONNECTED-COMPONENTS выполняет два поиска в глубину, имеется потенциальная неоднозначность при рассмотрении значений $u.d$ и $u.f$. В этом разделе данные значения всегда будут относиться ко времени открытия и времени завершения, вычисленным при первом вызове процедуры DFS в строке 1.

Мы распространим обозначения для времени открытия и времени завершения на множества вершин. Если $U \subseteq V$, то мы определим $d(U) = \min_{u \in U} \{u.d\}$ и $f(U) = \max_{u \in U} \{u.f\}$, т.е. $d(U)$ и $f(U)$ представляют собой самое раннее время открытия и самое позднее время завершения соответственно для всех вершин в U.

Следующая лемма и следствие из нее описывают ключевое свойство, связывающее сильно связные компоненты и времена завершения, полученные при первом поиске в глубину.

Лемма 22.14

Пусть C и C' – различные сильно связные компоненты в ориентированном графе $G = (V, E)$. Предположим, что имеется ребро $(u, v) \in E$, где $u \in C$ и $v \in C'$. Тогда $f(C) > f(C')$.

Доказательство. Имеется два возможных случая в зависимости от того, какой из сильно связных компонентов, C или C', содержит первую открытую в процессе поиска в глубину вершину.

Если $d(C) < d(C')$, то обозначим как x первую открытую в C вершину. В момент времени $x.d$ все вершины в C и C' – белые. В G имеется путь от x к каждой вершине в C, состоящий только из белых вершин. Поскольку $(u, v) \in E$, для любой вершины $w \in C'$ в момент времени $x.d$ в графе G имеется также путь от x к w, состоящий только из белых вершин: $x \leadsto u \to v \leadsto w$. Согласно теореме о белом пути все вершины в C и C' становятся потомками x в дереве поиска в глубину. Согласно следствию 22.8 x имеет самое позднее время завершения по сравнению со всеми его потомками, так что $x.f = f(C) > f(C')$.

Если же $d(C) > d(C')$, то обозначим как y первую открытую вершину в C'. В момент $y.d$ все вершины в C' белые, и в G имеется путь от y к каждой вершине C', состоящий только из белых вершин. В соответствии с теоремой о белом пути все вершины в C' становятся потомками y в дереве поиска в глубину, так что согласно следствию 22.8 $y.f = f(C')$. В момент времени $y.d$ все вершины в C белые. Поскольку имеется ребро (u, v) из C в C', из леммы 22.13 вытекает, что не существует пути из C' в C. Следовательно, в C не имеется вершин, достижимых из y. Таким образом, в момент времени $y.f$ все вершины в C остаются белыми. Значит, для любой вершины $w \in C$ имеем $w.f > y.f$, откуда следует, что $f(C) > f(C')$. ■

Приведенное ниже следствие говорит нам, что каждое ребро в G^{T}, соединяющее два сильно связных компонента, идет от компонента с более ранним временем завершения (при поиске в глубину) к компоненту с более поздним временем завершения.

Следствие 22.15

Пусть C и C' – различные сильно связные компоненты в ориентированном графе $G = (V, E)$. Предположим, что имеется ребро $(u, v) \in E^{\mathrm{T}}$, где $u \in C$ и $v \in C'$. Тогда $f(C) < f(C')$.

Доказательство. Поскольку $(u, v) \in E^{\mathrm{T}}$, мы имеем $(v, u) \in E$. Так как сильно связные компоненты G и G^{T} одни и те же, из леммы 22.14 следует, что $f(C) < f(C')$. ■

Следствие 22.15 дает нам ключ к пониманию того, почему работает процедура STRONGLY-CONNECTED-COMPONENTS. Рассмотрим, что происходит, когда мы выполняем второй поиск в глубину над графом G^{T}. Мы начинаем с сильно связного компонента C, время завершения которого $f(C)$ максимально. Поиск начинается с некоторой вершины $x \in C$, при этом посещаются все вершины в C. Согласно следствию 22.15 в G^{T} нет ребер из C в другой сильно связный компонент, так что при поиске из x не посещается ни одна вершина в других компонентах. Следовательно, дерево, корнем которого является x, содержит только вершины из C. После того как будут посещены все вершины в C, поиск в строке 3 выбирает в качестве корня вершину из некоторого другого сильно связного компонента C', время завершения $f(C')$ которого максимально среди всех компонентов, отличных от C. Теперь поиск посещает все вершины в C'. Согласно следствию 22.15 в G^{T} единственным ребром из C' в другие компоненты может быть ребро в C, но этот компонент уже посещен. В общем случае, когда поиск в глубину в G^{T} в строке 3 посещает некоторый сильно связный компонент, все ребра, исходящие из этого компонента, должны идти в уже обработанные компоненты. Следовательно, каждое дерево поиска в глубину является ровно одним сильно связным компонентом. Приведенная далее теорема формализует это доказательство.

Теорема 22.16

Процедура STRONGLY-CONNECTED-COMPONENTS(G) корректно вычисляет сильно связные компоненты ориентированного графа G.

Доказательство. Воспользуемся индукцией по количеству найденных деревьев при поиске в глубину в G^{T} в строке 3 и докажем, что вершины каждого дерева образуют сильно связный компонент. Гипотеза индукции состоит в том, что первые k деревьев, полученных в строке 3, являются сильно связными компонентами. Для базового случая $k = 0$ это утверждение тривиально.

Для выполнения шага индукции предположим, что каждое из первых k деревьев поиска в глубину в строке 3 представляет собой сильно связный компонент, и рассмотрим $(k + 1)$-е дерево. Пусть корнем этого дерева является вершина u и пусть u принадлежит сильно связному компоненту C. В соответствии с тем, как мы выбираем корни при поиске в глубину в строке 3, для любого сильно связного компонента C', который еще не был посещен и отличен от C, справедливо соотношение $u.f = f(C) > f(C')$. В соответствии с гипотезой индукции в момент времени, когда поиск посещает вершину u, все остальные вершины C — белые. Согласно теореме о белом пути все вершины C, кроме u, являются потомками u в дереве поиска в глубину. Кроме того, в соответствии с гипотезой индукции и со следствием 22.15 все ребра в G^{T}, которые покидают C, должны идти в уже посещенные сильно связные компоненты. Таким образом, ни в одном сильно связном

компоненте, отличном от C, нет вершины, которая бы стала потомком u в процессе поиска в глубину в G^{T}. Следовательно, вершины дерева поиска в глубину в G^{T}, корнем которого является u, образуют ровно один сильно связный компонент, что и завершает шаг индукции и доказательство данной теоремы. ■

Вот еще одна точка зрения на работу второго поиска в глубину. Рассмотрим граф компонентов $(G^{\mathrm{T}})^{\mathrm{SCC}}$ графа G^{T}. Если мы отобразим каждый сильно связный компонент, посещенный при втором поиске в глубину, на вершину $(G^{\mathrm{T}})^{\mathrm{SCC}}$, то вершины этого графа компонентов при втором поиске в глубину посещаются в порядке, обратном топологической сортировке. Если мы обратим все ребра графа $(G^{\mathrm{T}})^{\mathrm{SCC}}$, то получим граф $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Так как $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (см. упр. 22.5.4), при втором поиске в глубину вершины G^{SCC} посещаются в порядке топологической сортировки.

Упражнения

22.5.1

Как может измениться количество сильно связных компонентов графа при добавлении в граф нового ребра?

22.5.2

Покажите, как процедура STRONGLY-CONNECTED-COMPONENTS работает с графом, показанным на рис. 22.6. В частности, определите время завершения, вычисляемое в строке 1, и лес, полученный в строке 3. Считаем, что цикл в строках 5–7 процедуры DFS рассматривает вершины в алфавитном порядке и что так же упорядочены и списки смежности.

22.5.3

Профессор считает, что алгоритм определения сильно связных компонентов можно упростить, если во втором поиске в глубину использовать исходный, а не транспонированный граф, и сканировать вершины в порядке *возрастания* времени завершения. Всегда ли этот более простой алгоритм будет давать корректные результаты?

22.5.4

Докажите, что для любого ориентированного графа G справедливо соотношение $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$, т.е. что транспонирование графа компонентов для графа G^{T} дает тот же граф, что и граф компонентов графа G.

22.5.5

Разработайте алгоритм, который за время $O(V + E)$ находит граф компонентов ориентированного графа $G = (V, E)$. Убедитесь, что в полученном графе компонентов между двумя вершинами имеется не более одного ребра.

22.5.6

Поясните, как для данного ориентированного графа $G = (V, E)$ создать другой граф $G' = (V, E')$, такой, что: (а) G' имеет те же сильно связные компоненты, что

и G; (б) G' имеет тот же граф компонентов, что и G; (в) E' имеет минимально возможный размер. Разработайте быстрый алгоритм для вычисления G'.

22.5.7

Ориентированный граф $G = (V, E)$ называется ***полусвязным*** (semiconnected), если для всех пар вершин $u, v \in V$ мы имеем $u \leadsto v$ или $v \leadsto u$ (или и то, и другое одновременно). Разработайте эффективный алгоритм для определения, является ли данный граф G полусвязным. Докажите корректность разработанного алгоритма и проанализируйте время его работы.

Задачи

22.1. Классификация ребер при поиске в ширину

Лес поиска в глубину позволяет классифицировать ребра графа как ребра деревьев, обратные, прямые и перекрестные. Дерево поиска в ширину также можно использовать для аналогичной классификации ребер, достижимых из исходной вершины.

а. Докажите, что при поиске в ширину в неориентированном графе выполняются следующие свойства.

1. Не существует прямых и обратных ребер.
2. Для каждого ребра дерева (u, v) имеем $v.d = u.d + 1$.
3. Для каждого перекрестного ребра (u, v) имеем $v.d = u.d$ или $v.d = u.d+1$.

б. Докажите, что при поиске в ширину в ориентированном графе выполняются следующие свойства.

1. Не существует прямых ребер.
2. Для каждого ребра дерева (u, v) имеем $v.d = u.d + 1$.
3. Для каждого перекрестного ребра (u, v) имеем $v.d \leq u.d + 1$.
4. Для каждого обратного ребра (u, v) имеем $0 \leq v.d \leq u.d$.

22.2. Точки сочленения, мосты и двусвязные компоненты

Пусть $G = (V, E)$ является связным неориентированным графом. ***Точкой сочленения*** (articulation point) G называется вершина, удаление которой делает граф несвязным. ***Мостом*** (bridge) графа G называется ребро, удаление которого делает граф несвязным. ***Двусвязный компонент*** (biconnected component) графа G представляет собой максимальное множество ребер, такое что любые два ребра этого множества принадлежат общему простому циклу. На рис. 22.10 проиллюстрированы приведенные определения. Точки сочленения, мосты и двусвязные компоненты можно найти с помощью поиска в глубину. Пусть $G_\pi = (V, E_\pi)$ — дерево поиска в глубину графа G.

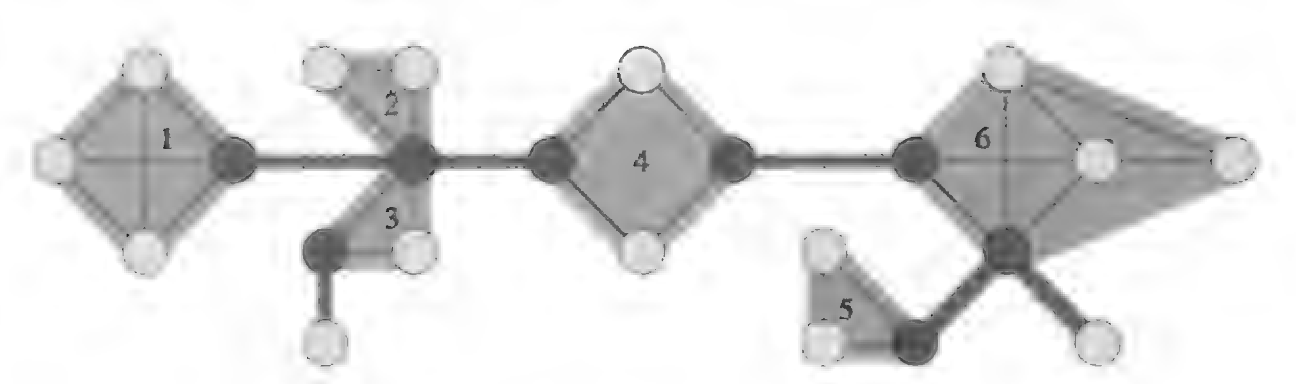


Рис. 22.10. Точки сочленения, мосты и двусвязные компоненты связного неориентированного графа для задачи 22.2. Темным цветом на рисунке выделены точки сочленения (вершины) и мосты (ребра), а двусвязные компоненты представляют собой наборы ребер в заштрихованных областях (внутри которых указаны номера *bcc*).

а. Докажите, что корень G_π — точка сочленения графа G тогда и только тогда, когда этот корень имеет как минимум два дочерних узла в G_π.

б. Пусть v — некорневая вершина G_π. Докажите, что v является точкой сочленения G тогда и только тогда, когда v имеет непосредственного потомка s, такого, что не существует обратного ребра от s или любого его потомка к истинному предку v.

в. Пусть $v.low$ — минимальное значение среди $v.d$ и всех $w.d$, где w — вершины, для которых имеется обратное ребро (u, w), где u — некоторый потомок вершины v. Покажите, как вычислить $v.low$ для всех вершин $v \in V$ за время $O(E)$.

г. Покажите, как найти все точки сочленения за время $O(E)$.

д. Докажите, что ребро в G является мостом тогда и только тогда, когда оно не принадлежит ни одному простому циклу G.

е. Покажите, как найти все мосты графа G за время $O(E)$.

ж. Докажите, что двусвязные компоненты графа G составляют разбиение множества всех ребер графа, не являющихся мостами.

з. Разработайте алгоритм, который за время $O(E)$ помечает каждое ребро e графа G натуральным числом $e.bcc$, таким, что $e.bcc = e'.bcc$ тогда и только тогда, когда e и e' находятся в одном и том же двусвязном компоненте.

22.3. Эйлеров цикл

Эйлеров цикл (Euler tour) сильно связного ориентированного графа $G = (V, E)$ представляет собой цикл, который проходит по всем ребрам G ровно по одному разу, хотя через вершины он может проходить по нескольку раз.

а. Покажите, что в G имеется эйлеров цикл тогда и только тогда, когда входящая степень каждой вершины равна ее исходящей степени: in-degree(v) = out-degree(v) для каждой вершины $v \in V$.

б. Разработайте алгоритм, который за время $O(E)$ находит эйлеров цикл графа G (если таковой цикл существует). (*Указание:* объединяйте циклы, у которых нет общих ребер.)

22.4. Достижимость

Пусть $G = (V, E)$ — ориентированный граф, в котором каждая вершина $u \in V$ помечена уникальным целым числом $L(u)$ из множества $\{1, 2, \ldots, |V|\}$. Для каждой вершины $u \in V$ рассмотрим множество $R(u) = \{v \in V : u \leadsto v\}$ вершин, достижимых из u. Определим $\min(u)$ как вершину в $R(u)$, метка которой минимальна, т.е. $\min(u)$ — это такая вершина v, что $L(v) = \min\{L(w) : w \in R(u)\}$. Разработайте алгоритм, который за время $O(V + E)$ вычисляет $\min(u)$ для всех вершин $u \in V$.

Заключительные замечания

Превосходные руководства по алгоритмам для работы с графами написаны Ивеном (Even) [102] и Таржаном (Tarjan) [328].

Поиск в ширину был открыт Муром (Moore) [258] в контексте задачи поиска пути через лабиринт. Ли (Lee) [225] независимо открыл тот же алгоритм при работе над разводкой печатных плат.

Хопкрофт (Hopcroft) и Таржан [177] указали на преимущества использования представления графов в виде списков смежности над матричным представлением для разреженных графов и были первыми, кто оценил алгоритмическую важность поиска в глубину. Поиск в глубину широко используется с конца 1950-х годов, в особенности в программах искусственного интеллекта.

Таржан [325] разработал алгоритм поиска сильно связных компонентов за линейное время. Алгоритм из раздела 22.5 взят у Ахо (Aho), Хопкрофта и Ульмана (Ullman) [6], которые ссылаются на неопубликованную работу С.Р. Косараю (S.R. Kosaraju) и работу Шарира (Sharir) [312]. Габов (Gabow) [118] разработал алгоритм для поиска сильно связных компонентов, который основан на сжатых циклах и использует два стека для обеспечения линейного времени работы. Кнут (Knuth) [208][4] первым разработал алгоритм топологической сортировки за линейное время.

[4]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 23. Минимальные остовные деревья

При разработке электронных схем зачастую необходимо электрически соединить контакты нескольких компонентов. Для соединения множества из n контактов можно использовать некоторую компоновку из $n - 1$ проводов, каждый из которых соединяет два контакта. Обычно желательно получить компоновку, которая использует минимальное количество провода.

Эту задачу можно смоделировать с помощью связного неориентированного графа $G = (V, E)$, где V — множество контактов, E — множество возможных соединений между парами контактов, и для каждого ребра $(u, v) \in E$ задан вес $w(u, v)$, определяющий стоимость (количество необходимого провода) соединения u и v. Мы хотим найти ациклическое подмножество $T \subseteq E$, которое соединяет все вершины и общий вес которого

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

минимален. Поскольку множество T ациклическое и связывает все вершины, оно должно образовывать дерево, которое мы назовем ***остовным деревом*** (spanning tree) графа G. Задача поиска дерева T называется ***задачей поиска минимального остовного дерева*** (minimum-spanning-tree problem)[1]. На рис. 23.1 показан пример связного графа и его минимального остовного дерева.

В этой главе мы рассмотрим два алгоритма решения задачи поиска минимального остовного дерева — алгоритмы Крускала (Kruskal) и Прима (Prim). Каждый из них легко реализовать с помощью обычных бинарных пирамид, получив время работы $O(E \lg V)$. При использовании фибоначчиевых пирамид алгоритм Прима можно ускорить до $O(E + V \lg V)$, что является весьма существенным ускорением, когда $|V|$ гораздо меньше $|E|$.

Оба эти алгоритма — жадные (см. главу 16). На каждом шаге алгоритма мы выбираем один из возможных вариантов. Жадная стратегия предполагает выбор варианта, наилучшего в данный момент. В общем случае такая стратегия не га-

[1]По сути, термин "минимальное остовное дерево" означает "остовное дерево с минимальным весом". Мы не минимизируем, например, количество ребер в T, поскольку все остовные деревья имеют ровно $|V| - 1$ ребер согласно теореме Б.2.

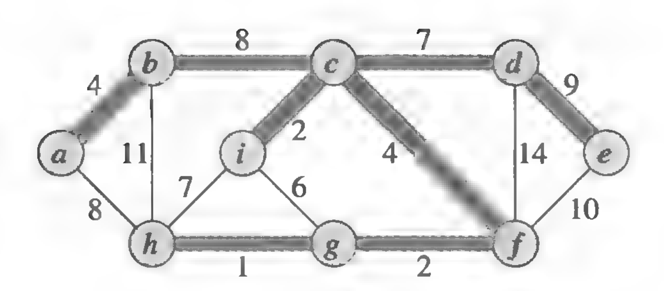


Рис. 23.1. Минимальное остовное дерево связного графа. На ребрах указан их вес, а ребра минимального остовного дерева отдельно выделены цветом. Общий вес показанного дерева равен 37. Приведенное минимальное дерево не единственное: удалив ребро (b, c) и заменив его ребром (a, h), мы получим другое остовное дерево с тем же весом 37.

рантирует глобально оптимального решения задачи, однако для задачи поиска минимального остовного дерева можно доказать, что определенные жадные стратегии дают остовное дерево минимального веса. Хотя настоящую главу можно читать независимо от главы 16, жадные алгоритмы, представленные здесь, наглядно демонстрируют классическое применение изложенных в этой главе теоретических основ.

В разделе 23.1 описан "обобщенный" метод построения минимального остовного дерева, который наращивает остовное дерево по одному ребру. В разделе 23.2 приведены два варианта реализации обобщенного алгоритма. Первый алгоритм (Крускала) похож на алгоритм поиска связных компонентов из раздела 21.1. Второй алгоритм (Прима) подобен алгоритму поиска кратчайшего пути Дейкстры (Dijkstra) из раздела 24.3.

Поскольку дерево является разновидностью графа, чтобы быть точными, мы должны определять дерево не только через его ребра, но и через вершины. Хотя в этой главе деревья рассматриваются в основном через вершины, следует понимать, что вершины дерева T — это то, что соединяют ребра дерева T.

23.1. Выращивание минимального остовного дерева

Предположим, что у нас есть связный неориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ и мы хотим найти минимальное остовное дерево для G. В этой главе мы рассмотрим два алгоритма решения поставленной задачи, использующих, хотя и по-разному, один и тот же жадный подход.

Эта жадная стратегия используется следующим обобщенным методом, который "выращивает" минимальное остовное дерево по одному ребру. Обобщенный метод работает с множеством ребер A, поддерживая следующий инвариант цикла.

> Перед каждой очередной итерацией A представляет собой подмножество некоторого минимального остовного дерева.

На каждом шаге алгоритма мы определяем ребро (u, v), которое можно добавить к A без нарушения этого инварианта, в том смысле, что $A \cup \{(u, v)\}$ также является подмножеством минимального остовного дерева. Мы назовем такое ребро

безопасным (safe edge) для A, поскольку его можно добавить к A, не опасаясь нарушить инвариант.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A не образует остовного дерева
3      Найти ребро (u, v), безопасное для A
4      A = A ∪ {(u, v)}
5  return A
```

Инвариант цикла используется следующим образом.

Инициализация. После выполнения строки 1 множество A тривиально удовлетворяет инварианту цикла.

Сохранение. Цикл в строках 2–4 сохраняет инвариант путем добавления только безопасных ребер.

Завершение. Все ребра, добавленные в A, находятся в минимальном остовном дереве, так что множество A, возвращаемое в строке 5, должно быть минимальным остовным деревом.

Самое сложное, само собой разумеется, заключается в том, как именно найти безопасное ребро в строке 3. Оно должно существовать, поскольку, когда выполняется строка 3, инвариант требует, чтобы существовало такое остовное дерево T, что $A \subseteq T$. Внутри тела цикла **while** A должно быть истинным подмножеством T, поэтому должно существовать ребро $(u, v) \in T$, такое что $(u, v) \notin A$ и (u, v) — безопасное для A ребро.

В оставшейся части этого раздела мы приведем правило (теорема 23 1) для распознавания безопасных ребер. В следующем разделе описаны два алгоритма, которые используют это правило для эффективного поиска безопасных ребер.

Сначала нам потребуется несколько определений. ***Разрезом*** (cut) $(S, V - S)$ неориентированного графа $G = (V, E)$ называется разбиение V, что проиллюстрировано на рис. 23.2. Мы говорим, что ребро $(u, v) \in E$ ***пересекает*** (crosses) разрез $(S, V - S)$, если один из его концов оказывается в множестве S, а второй — в $V - S$. Разрез ***согласован*** (respect) с множеством ребер A, если ни одно ребро из A не пересекает разрез. Ребро, пересекающее разрез, является ***легким*** (light), если оно имеет минимальный вес среди всех ребер, пересекающих разрез. Заметим, что может быть несколько легких ребер одновременно. В общем случае мы говорим, что ребро является ***легким ребром***, удовлетворяющим некоторому свойству, если оно имеет минимальный вес среди всех ребер, удовлетворяющих этому свойству.

Правило для распознавания безопасных ребер дает следующая теорема.

Теорема 23.1

Пусть $G = (V, E)$ является связным неориентированным графом с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, кото-

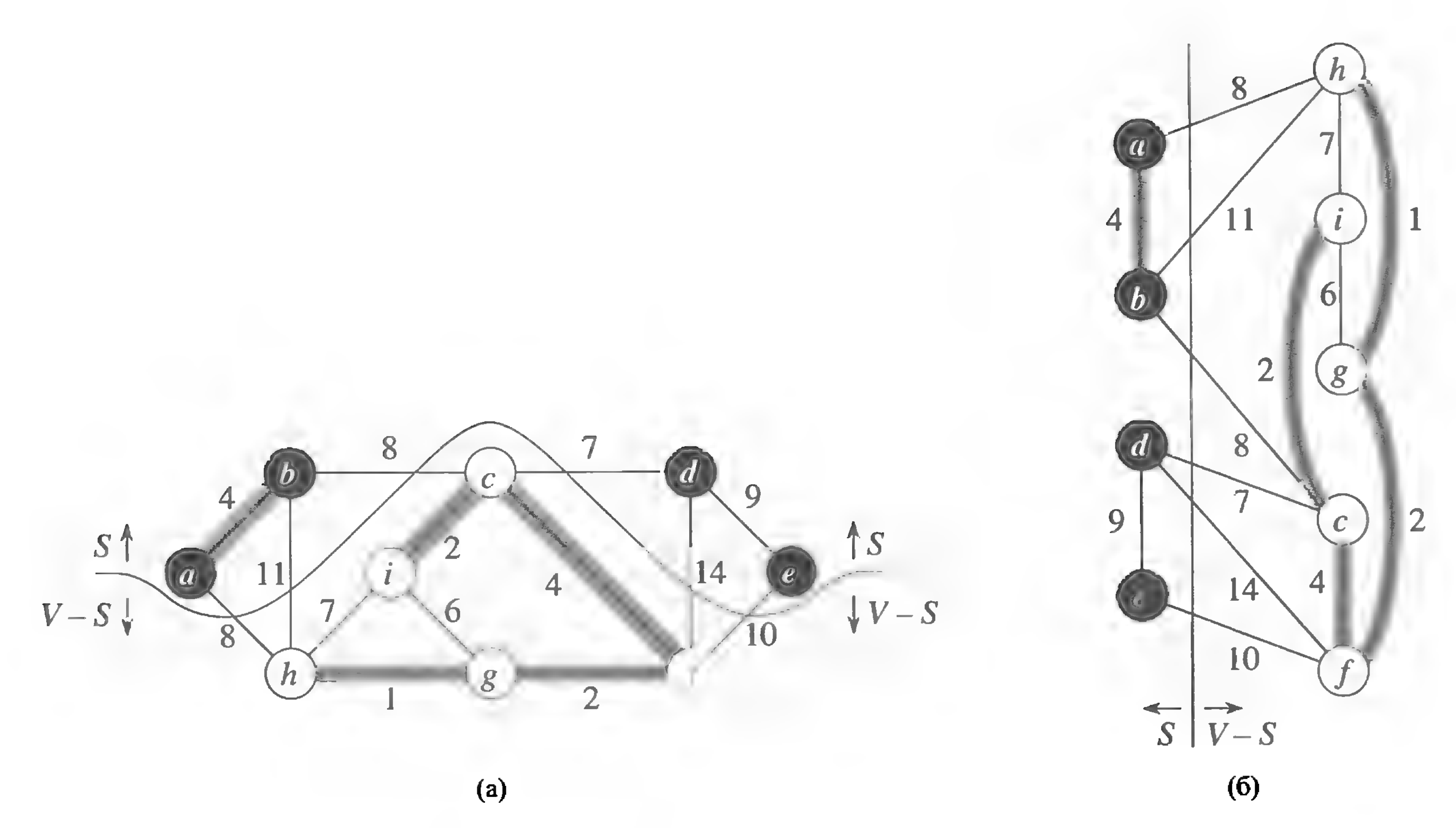


Рис. 23.2. Два варианта представления разреза $(S, V-S)$ графа, показанного на рис. 23.1. **(а)** Черные вершины находятся в множестве S, а белые — в $V-S$. Ребра, пересекающие разрез, соединяют черные и белые вершины. Ребро (d, c) является единственным легким ребром, пересекающим разрез. Подмножество ребер A заштриховано; заметим, что разрез $(S, V-S)$ согласован с множеством A, поскольку нет ни одного ребра из A, которое пересекало бы разрез. **(б)** Тот же граф с вершинами из множества S слева и вершинами из множества $V-S$ справа. Ребро пересекает разрез, если оно соединяет вершину слева с вершиной справа.

рое входит в некоторое минимальное остовное дерево G, $(S, V-S)$ — любой разрез G, согласованный с A, а (u, v) — легкое ребро, пересекающее разрез $(S, V-S)$. Тогда ребро (u, v) является безопасным для A.

Доказательство. Пусть T — минимальное остовное дерево, которое включает A, и предположим, что T не содержит ребро (u, v), поскольку в противном случае теорема доказана. Мы построим другое минимальное остовное дерево T', которое включает $A \cup \{(u, v)\}$, путем использования метода вырезания и вставки, показывая таким образом, что ребро (u, v) является безопасным для A.

Ребро (u, v) образует цикл с ребрами на простом пути p от u до v в T, как показано на рис. 23.3. Поскольку u и v находятся на разных сторонах разреза $(S, V-S)$, на пути p имеется как минимум одно ребро из T, которое пересекает разрез. Пусть таким ребром является ребро (x, y). Оно не входит в A, поскольку разрез согласован с A. Так как (x, y) лежит на единственном пути от u к v в T, его удаление разбивает T на два компонента. Добавление (u, v) восстанавливает разбиение, образуя новое остовное дерево $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Теперь покажем, что T' — минимальное остовное дерево. Поскольку (u, v) — легкое ребро, пересекающее разбиение $(S, V-S)$, и (x, y) также пересекает это

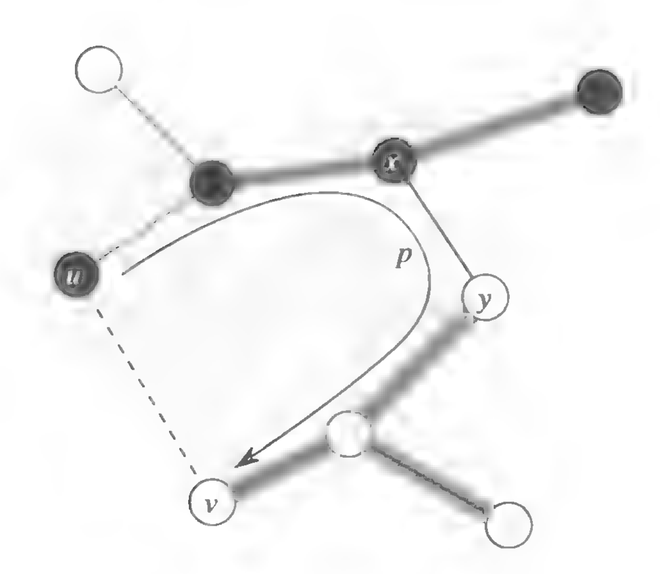


Рис. 23.3. Доказательство теоремы 23.1. Черные вершины находятся в S, а белые — в $V - S$. Показаны ребра в минимальном остовном дереве T, но не ребра графа G. Ребра в A заштрихованы, а (u, v) представляет собой легкое ребро, пересекающее разрез $(S, V - S)$. Ребро (x, y) является ребром на единственном простом пути p от u до v в T. Чтобы получить минимальное остовное дерево T', которое содержит (u, v), нужно удалить из T ребро (x, y) и добавить ребро (u, v).

разбиение, $w(u, v) \leq w(x, y)$. Следовательно,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) . \end{aligned}$$

Однако T — минимальное остовное дерево, так что $w(T) \leq w(T')$. Следовательно, T' также должно быть минимальным остовным деревом.

Остается показать, что (u, v) действительно безопасное ребро для A. Мы имеем $A \subseteq T'$, поскольку $A \subseteq T$ и $(x, y) \notin A$. Таким образом, $A \cup \{(u, v)\} \subseteq T'$ и, поскольку T' — минимальное остовное дерево, ребро (u, v) безопасно для A. ∎

Теорема 23.1 помогает лучше понять, как метод GENERIC-MST работает со связным графом. В процессе работы алгоритма множество A всегда ациклическое; в противном случае минимальное остовное дерево, включающее A, содержало бы цикл, что приводит к противоречию. В любой момент выполнения алгоритма граф $G_A = (V, A)$ представляет собой лес, а каждый из связных компонентов G_A является деревом. (Некоторые из деревьев могут содержать всего одну вершину, например, в случае, когда алгоритм начинает работу: множество A в этот момент пустое, а лес содержит $|V|$ деревьев, по одному для каждой вершины.) Кроме того, любое безопасное для A ребро (u, v) соединяет различные компоненты G_A, поскольку множество $A \cup \{(u, v)\}$ должно быть ациклическим.

Цикл **while** в строках 2–4 процедуры GENERIC-MST выполняется $|V| - 1$ раз, он находит по одному из $|V| - 1$ ребер минимального остовного дерева на каждой итерации. Изначально, когда $A = \emptyset$, в G_A имеется $|V|$ деревьев, и каждая итерация уменьшает их количество на единицу. Когда лес состоит только из одного дерева, алгоритм завершается.

Два алгоритма, рассмотренные в разделе 23.2, используют следствие из теоремы 23.1.

Следствие 23.2

Пусть $G = (V, E)$ — связный неориентированный граф с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, которое входит в некоторое минимальное остовное дерево G, и пусть $C = (V_C, E_C)$ — связный компонент (дерево) в лесу $G_A = (V, A)$. Если (u, v) является легким ребром, соединяющим C с некоторым другим компонентом в G_A, то ребро (u, v) безопасно для A.

Доказательство. Разрез $(V_C, V - V_C)$ согласован с A, а (u, v) — легкое ребро для данного разреза. Следовательно, ребро (u, v) безопасно для A. ■

Упражнения

23.1.1

Пусть (u, v) — ребро минимального веса в связном графе G. Покажите, что (u, v) принадлежит некоторому минимальному остовному дереву G.

23.1.2

Профессор утверждает, что верно следующее обращение теоремы 23.1. Пусть $G = (V, E)$ — связный неориентированный граф с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, входящее в некоторое минимальное остовное дерево G, $(S, V - S)$ — произвольный разрез G, согласованный с A, и пусть (u, v) — безопасное для A ребро, пересекающее разрез $(S, V - S)$. Тогда (u, v) — легкое ребро для данного разреза. Приведите контрпример, демонстрирующий некорректность профессорского обращения теоремы.

23.1.3

Покажите, что если ребро (u, v) содержится в некотором минимальном остовном дереве, то оно является легким ребром, пересекающим некоторый разрез графа.

23.1.4

Приведите простой пример связного графа, такого, что множество ребер $\{(u, v) :$ существует разрез $(S, V - S)$, такой, что (u, v) является легким ребром, пересекающим $(S, V - S)\}$ не образует минимального остовного дерева.

23.1.5

Пусть e — ребро с максимальным весом в некотором цикле связного графа $G = (V, E)$. Докажите, что имеется минимальное остовное дерево графа $G' = (V, E - \{e\})$, которое одновременно является минимальным остовным деревом G, т.е. что существует минимальное остовное дерево G, не включающее e.

23.1.6

Покажите, что граф имеет единственное минимальное остовное дерево, если для каждого разреза графа имеется единственное легкое ребро, пересекающее этот разрез. Покажите с помощью контрпримера, что обратное утверждение не верно.

23.1.7

Покажите, что если вес любого из ребер графа положителен, то любое подмножество ребер, объединяющее все вершины и имеющее минимальный общий вес, должно быть деревом. Приведите пример, показывающий, что это не так, если ребра могут иметь отрицательный вес.

23.1.8

Пусть T — минимальное остовное дерево графа G и пусть L — отсортированный список весов ребер T. Покажите, что для любого другого минимального остовного дерева T' графа G отсортированный список весов ребер будет тем же.

23.1.9

Пусть T — минимальное остовное дерево графа $G = (V, E)$, а V' — подмножество V. Пусть T' — подграф T, порожденный V', а G' — подграф G, порожденный V'. Покажите, что если T' — связный граф, то он является минимальным остовным деревом G'.

23.1.10

Пусть даны граф G и минимальное остовное дерево T. Предположим, что мы уменьшаем вес одного из ребер в T. Покажите, что T при этом останется минимальным остовным деревом G. Говоря более строго, пусть T — минимальное остовное дерево G с весами ребер, заданными весовой функцией w. Выберем одно ребро $(x, y) \in T$ и положительное число k и определим весовую функцию w' следующим образом:

$$w'(u,v) = \begin{cases} w(u,v) \ , & \text{если } (u,v) \neq (x,y) \ , \\ w(x,y) - k \ , & \text{если } (u,v) = (x,y) \ . \end{cases}$$

Покажите, что если веса ребер определяются функцией w', то T остается минимальным остовным деревом G.

23.1.11 ★

Пусть задан граф G и минимальное остовное дерево T. Предположим, что мы уменьшаем вес одного из ребер, не входящих в T. Разработайте алгоритм для поиска минимального остовного дерева модифицированного графа.

23.2. Алгоритмы Крускала и Прима

Два описанных в этом разделе алгоритма следуют общей схеме поиска минимального остовного дерева. Каждый из них использует свое правило для определения безопасных ребер в строке 3 процедуры GENERIC-MST. В алгоритме Крускала множество A является лесом. В A добавляются безопасные ребра, которые являются ребрами минимального веса, объединяющими два различных компонента. В алгоритме Прима множество A образует единое дерево. В A добавляются

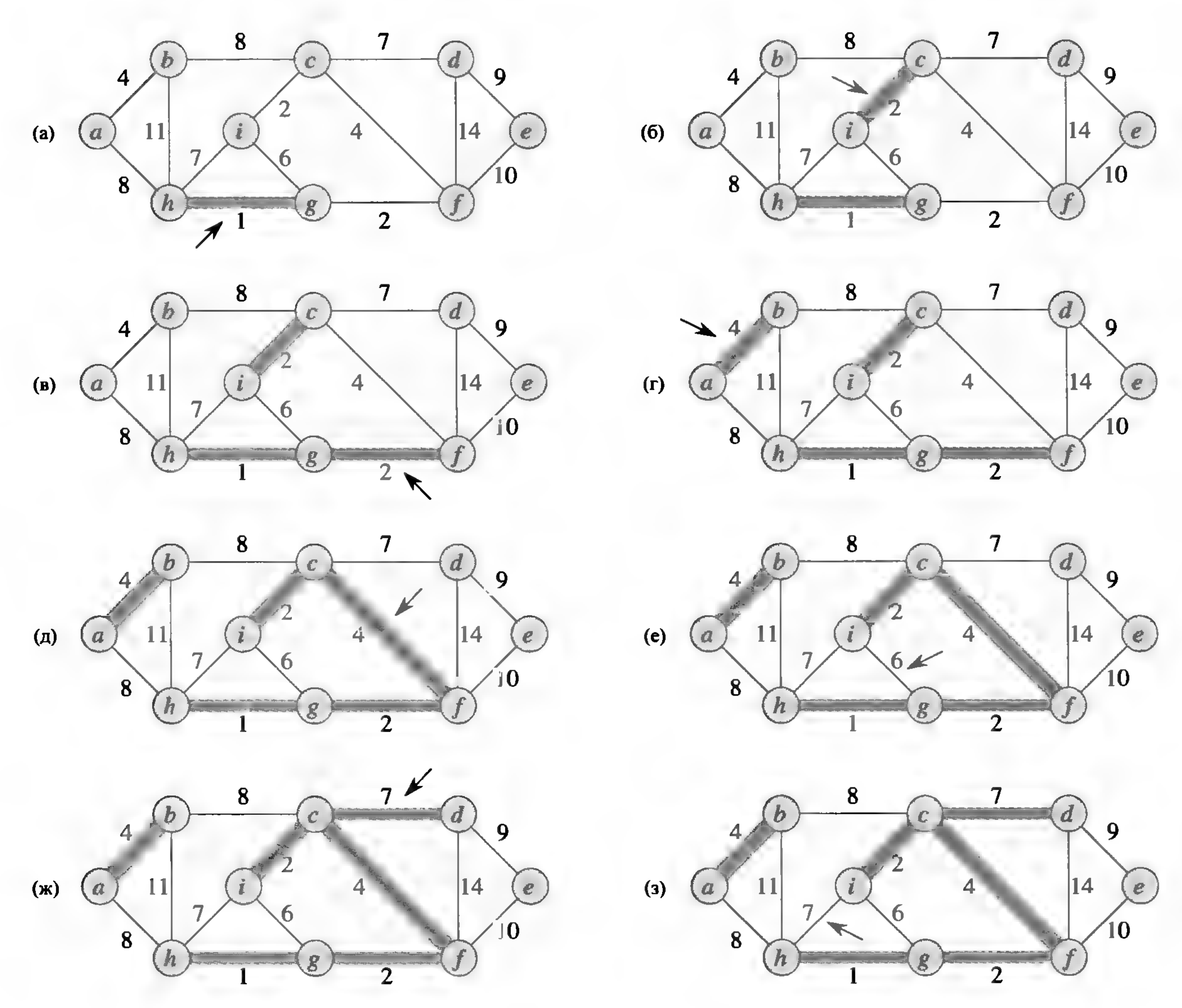


Рис. 23.4. Применение алгоритма Крускала к графу, показанному на рис. 23.1. Заштрихованные ребра принадлежат растущему лесу A. Алгоритм рассматривает ребра в порядке возрастания их веса. На каждом шаге алгоритма стрелка указывает на рассматриваемое ребро. Если ребро объединяет два различных дерева леса, оно добавляется в лес, тем самым сливая эти деревья в одно.

безопасные ребра, которые являются ребрами минимального веса, соединяющими дерево с вершиной вне дерева.

Алгоритм Крускала

Алгоритм Крускала находит безопасное ребро для добавления в растущий лес путем поиска ребра (u, v) с минимальным весом среди всех ребер, соединяющих два дерева в лесу. Обозначим два дерева, соединяемые ребром (u, v), как C_1 и C_2. Поскольку (u, v) должно быть легким ребром, соединяющим C_1 с некоторым другим деревом, из следствия 23.2 вытекает, что (u, v) — безопасное для C_1 ребро. Алгоритм Крускала является жадным, поскольку на каждом шаге он добавляет к лесу ребро с минимально возможным весом.

Наша реализация алгоритма Крускала похожа на алгоритм для вычисления связных компонентов из раздела 21.1. Она использует структуру для представления непересекающихся множеств. Каждое из множеств содержит вершины неко-

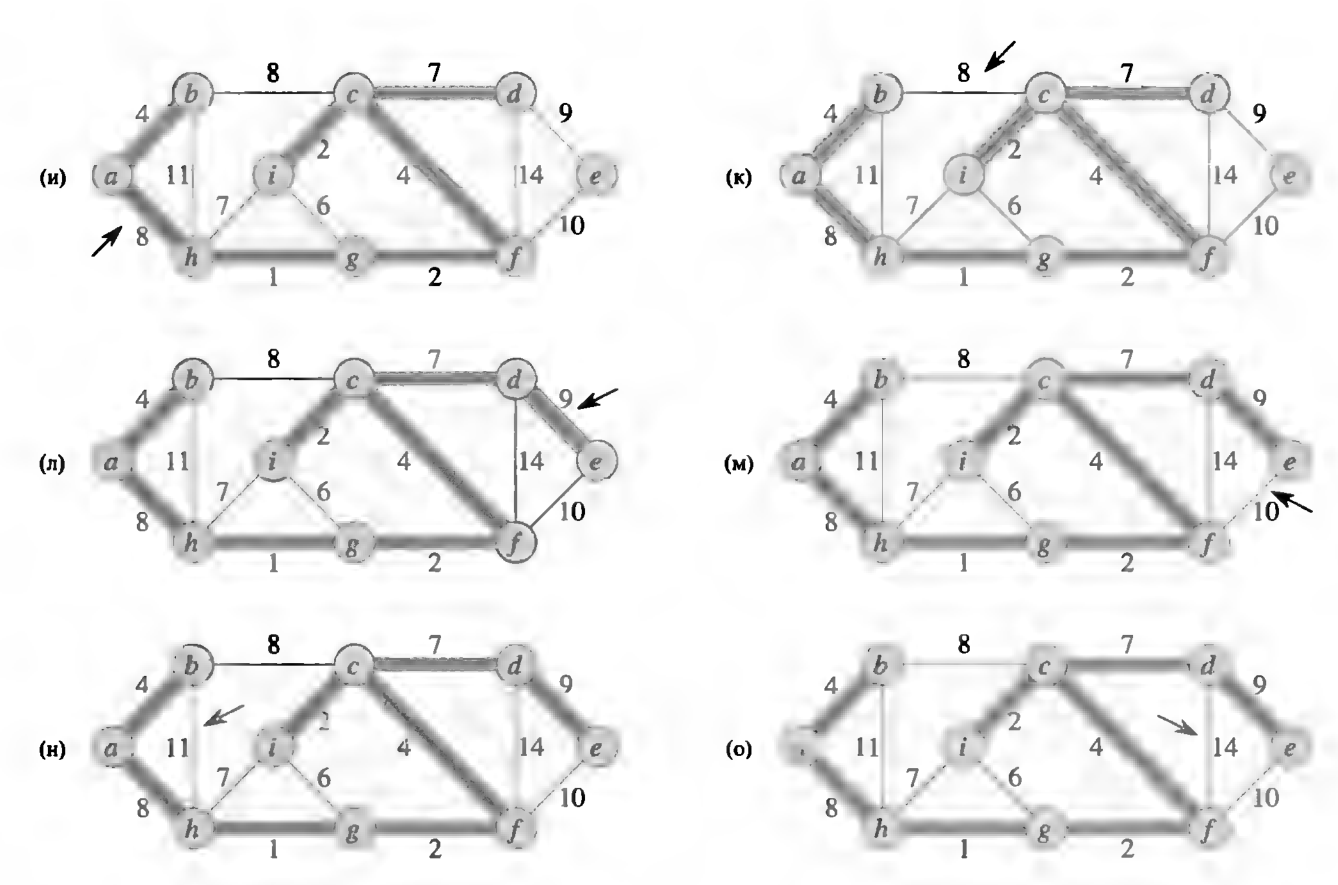


Рис. 23.4 (продолжение). Завершение выполнения алгоритма Крускала.

торого дерева в текущем лесу. Операция FIND-SET(u) возвращает представитель множества, содержащего u. Таким образом, мы можем определить, принадлежат ли вершины u и v одному и тому же дереву, проверив равенство FIND-SET(u) и FIND-SET(v). Объединение деревьев выполняется с помощью процедуры UNION.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for каждой вершины v ∈ G.V
3      MAKE-SET(v)
4  Отсортировать ребра G.E в неуменьшающемся порядке по весу w
5  for каждого ребра (u, v) ∈ G.E в этом порядке
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Работа алгоритма Крускала показана на рис. 23.4. В строках 1–3 выполняется инициализация множества A пустым множеством и создаются $|V|$ деревьев, каждое из которых содержит по одной вершине. В строке 4 ребра в E сортируются согласно их весу в неубывающем порядке. Цикл **for** в строках 5–8 проверяет для каждого ребра (u, v) в указанном неубывающем по весу порядке, принадлежат ли его концы u и v одному и тому же дереву. Если это так, то данное ребро не может быть добавлено к лесу без того, чтобы создать при этом цикл, поэтому в таком

случае ребро отбрасывается. В противном случае, когда концы ребра принадлежат разным деревьям, в строке 7 ребро (u, v) добавляется в множество A, и вершины двух деревьев объединяются в строке 8.

Время работы алгоритма Крускала для графа $G = (V, E)$ зависит от реализации структуры данных для непересекающихся множеств. Мы будем считать, что лес непересекающихся множеств реализован так, как описано в разделе 21.3, с эвристиками объединения по рангу и сжатия пути, поскольку асимптотически это наиболее быстрая известная реализация. Инициализация множества A в строке 1 занимает время $O(1)$, а время, необходимое для сортировки множества в строке 4, равно $O(E \lg E)$ (стоимость $|V|$ операций MAKE-SET в цикле **for** в строках 2 и 3 мы учтем чуть позже). Цикл **for** в строках 5–8 выполняет $O(E)$ операций FIND-SET и UNION над лесом непересекающихся множеств. Вместе с $|V|$ операциями MAKE-SET эта работа требует времени $O((V + E)\,\alpha(V))$, где α — очень медленно растущая функция, определенная в разделе 21.4. Поскольку мы предполагаем, что G — связный граф, справедливо соотношение $|E| \geq |V| - 1$, так что операции над непересекающимися множествами требуют времени $O(E\,\alpha(V))$. Кроме того, поскольку $\alpha(|V|) = O(\lg V) = O(\lg E)$, общее время работы алгоритма Крускала равно $O(E \lg E)$. Заметим, что $|E| < |V|^2$, поэтому $\lg |E| = O(\lg V)$ и время работы алгоритма Крускала можно записать как $O(E \lg V)$.

Алгоритм Прима

Как и алгоритм Крускала, алгоритм Прима представляет собой частный случай обобщенного алгоритма поиска минимального остовного дерева из раздела 23.1. Алгоритм Прима очень похож на алгоритм Дейкстры для поиска кратчайшего пути в графе, который будет рассмотрен в разделе 24.3. Алгоритм Прима обладает тем свойством, что ребра в множестве A всегда образуют единое дерево. Как показано на рис. 23.5, дерево начинается с произвольной корневой вершины r и растет до тех пор, пока не охватит все вершины в V. На каждом шаге к дереву A добавляется легкое ребро, соединяющее дерево и отдельную вершину из оставшейся части графа. Согласно следствию 23.2 данное правило добавляет только безопасные для A ребра; следовательно, по завершении алгоритма ребра в A образуют минимальное остовное дерево. Данная стратегия является жадной, поскольку на каждом шаге к дереву добавляется ребро, которое вносит минимально возможный вклад в общий вес.

Для эффективной реализации алгоритма Прима необходим быстрый способ выбора нового ребра для добавления в дерево. В приведенном ниже псевдокоде в качестве входных данных алгоритму передаются связный граф G и корень r минимального остовного дерева, которое будет "выращено" алгоритмом. В процессе работы алгоритма все вершины, которые *не* входят в дерево, располагаются в невозрастающей очереди с приоритетами Q, основанной на значении атрибута key. Для каждой вершины v значение атрибута $v.key$ представляет собой минимальный вес среди всех ребер, соединяющих v с вершиной в дереве. Если ни одного такого ребра нет, считаем, что $v.key = \infty$. Атрибут $v.\pi$ указывает родителя v в дереве. В процессе работы алгоритма множество A из процедуры

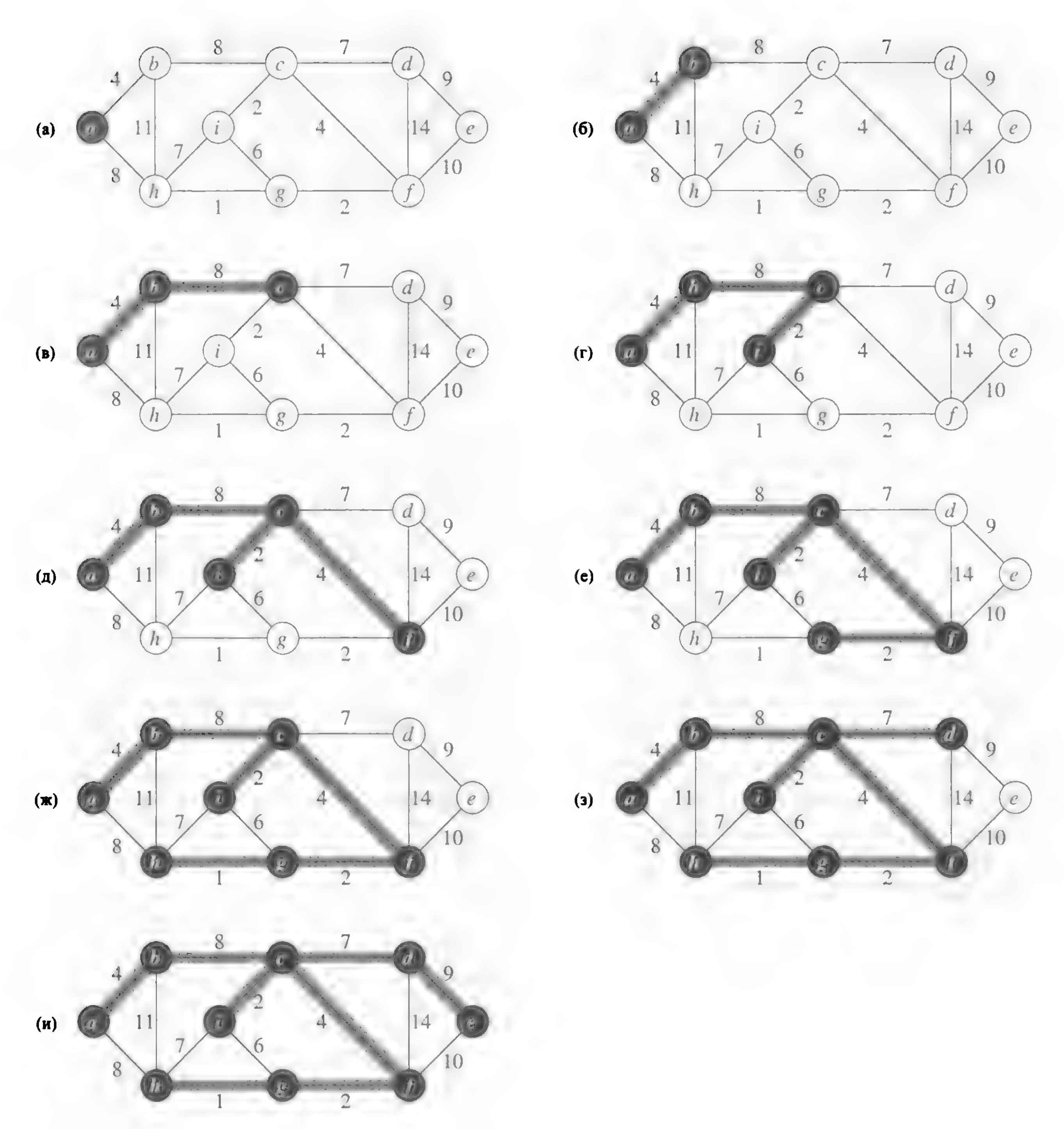


Рис. 23.5. Применение алгоритма Прима к графу, показанному на рис. 23.1. Корневой вершиной является вершина a. Заштрихованные ребра принадлежат растущему дереву; черным цветом показаны вершины, находящиеся в этом дереве. На каждой итерации алгоритма вершины дерева определяют разрез графа, и к дереву добавляется легкое ребро, пересекающее разрез. Например, на второй итерации алгоритм может выбрать либо ребро (b, c), либо ребро (a, h), поскольку оба они являются легкими ребрами, пересекающими разрез.

GENERIC-MST неявно поддерживается как

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\} \ .$$

Когда алгоритм завершает работу, очередь с приоритетами Q пуста и минимальным остовным деревом для G является дерево

$$A = \{(v, v.\pi) : v \in V - \{r\}\} \ .$$

```
MST-PRIM(G, w, r)
 1  for каждой вершины u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for каждой вершины v ∈ G.Adj[u]
 9          if v ∈ Q и w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
                // С вызовом DECREASE-KEY(Q, v, w(u, v))
```

Работа алгоритма Прима проиллюстрирована на рис. 23.5. В строках 1–5 ключи всех вершин устанавливаются равными ∞ (кроме корня r, ключ которого равен 0, так что он оказывается первой обрабатываемой вершиной), указателям на родителей для всех узлов присваиваются значения NIL и все вершины вносятся в неубывающую очередь с приоритетами Q. Алгоритм поддерживает следующий инвариант цикла, состоящий из трех частей.

Перед каждой итерацией цикла **while** в строках 6–11

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$;
2. вершины, уже помещенные в минимальное остовное дерево, принадлежат множеству $V - Q$;
3. для всех вершин $v \in Q$ справедливо следующее: если $v.\pi \neq$ NIL, то $v.key < \infty$ и $v.key$ — вес легкого ребра $(v, v.\pi)$, соединяющего v с некоторой вершиной, уже находящейся в минимальном остовном дереве.

В строке 7 определяется вершина $u \in Q$, инцидентная легкому ребру, пересекающему разрез $(V - Q, Q)$ (за исключением первой итерации, когда $u = r$ в соответствии с присвоением в строке 4). Удаление u из множества Q добавляет ее в множество $V - Q$ вершин дерева, таким образом добавляя $(u, u.\pi)$ в A. Цикл **for** в строках 8–11 обновляет атрибуты key и π каждой вершины v, смежной с u и не находящейся в дереве. Это обновление сохраняет третью часть инварианта.

Время работы алгоритма Прима зависит от выбранной реализации невозрастающей очереди с приоритетами Q. Если реализовать ее как бинарную пирамиду

(см. главу 6), то для выполнения инициализации в строках 1–5 за время $O(V)$ можно использовать процедуру BUILD-MIN-HEAP. Тело цикла **while** выполняется $|V|$ раз, а поскольку каждая операция EXTRACT-MIN занимает время $O(\lg V)$, общее время всех вызовов процедур EXTRACT-MIN составляет $O(V \lg V)$. Цикл **for** в строках 8–11 выполняется всего $O(E)$ раз, поскольку сумма длин всех списков смежности равна $2|E|$. Внутри цикла **for** проверка на принадлежность Q в строке 9 может быть реализована за постоянное время, если воспользоваться для каждой вершины битом, указывающим, находится ли она в Q, и обновлять этот бит при удалении вершины из Q. Присвоение в строке 11 неявно включает операцию DECREASE-KEY над пирамидой. Время выполнения этой операции в бинарной пирамиде — $O(\lg V)$. Таким образом, общее время работы алгоритма Прима составляет $O(V \lg V + E \lg V) = O(E \lg V)$, что асимптотически совпадает со временем работы рассмотренной ранее реализации алгоритма Крускала.

Асимптотическое время работы алгоритма Прима можно улучшить за счет применения фибоначчиевых пирамид. В главе 19 показано, что если $|V|$ элементов организованы в фибоначчиеву пирамиду, то операцию EXCTARCT-MIN можно выполнить за амортизированное время $O(\lg V)$, а операцию DECREASE-KEY (для реализации строки 11) — за амортизированное время $O(1)$. Следовательно, при использовании фибоначчиевой пирамиды для реализации неубывающей очереди с приоритетами Q общее время работы алгоритма Прима улучшается до $O(E + V \lg V)$.

Упражнения

23.2.1

Алгоритм Крускала может возвращать разные остовные деревья для одного и того же входного графа G в зависимости от взаимного расположения ребер с одинаковым весом при сортировке. Покажите, что для любого минимального остовного дерева T графа G можно указать способ сортировки ребер G, для которого алгоритм Крускала даст минимальное остовное дерево T.

23.2.2

Предположим, что граф $G = (V, E)$ представлен с помощью матрицы смежности. Разработайте для этого случая простую реализацию алгоритма Прима, время работы которой равно $O(V^2)$.

23.2.3

Будет ли реализация алгоритма Прима с использованием фибоначчиевых пирамид асимптотически быстрее реализации с использованием бинарных пирамид для разреженного графа $G = (V, E)$, для которого $|E| = \Theta(V)$? А для плотного графа, в котором $|E| = \Theta(V^2)$? Каким образом должны быть связаны $|E|$ и $|V|$, чтобы реализация с использованием фибоначчиевых пирамид была быстрее реализации с использованием бинарных пирамид?

22 Зак. 3726

23.2.4

Предположим, что все веса ребер графа представляют собой целые числа в диапазоне от 1 до $|V|$. Насколько быстрым можно сделать алгоритм Крускала в этом случае? А в случае, когда вес каждого ребра представляет собой целое число в диапазоне от 1 до W для некоторой константы W?

23.2.5

Предположим, что все веса ребер графа представляют собой целые числа в диапазоне от 1 до $|V|$. Насколько быстрым можно сделать алгоритм Прима в этом случае? А в случае, когда вес каждого ребра представляет собой целое число в диапазоне от 1 до W для некоторой константы W?

23.2.6 ⋆

Предположим, что веса ребер графа равномерно распределены на полуоткрытом интервале $[0, 1)$. Какой алгоритм — Крускала или Прима — будет работать быстрее в этом случае?

23.2.7 ⋆

Предположим, что граф G имеет уже вычисленное минимальное остовное дерево. Насколько быстро можно обновить минимальное остовное дерево при добавлении в G новой вершины и инцидентных ребер?

23.2.8

Профессор предложил новый алгоритм декомпозиции для вычисления минимальных остовных деревьев, заключающийся в следующем. Для данного графа $G = (V, E)$ разбиваем множество вершин V на два подмножества V_1 и V_2, таких, что $|V_1|$ и $|V_2|$ отличаются не более чем на 1. Пусть E_1 — множество ребер, инцидентных только с вершинами в V_1, а E_2 — множество ребер, инцидентных только с вершинами в V_2. Рекурсивно решаем задачу поиска минимальных остовных деревьев в каждом из подграфов $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$, а затем выбираем среди ребер E ребро с минимальным весом, пересекающее разрез (V_1, V_2), и используем его для объединения двух полученных минимальных остовных деревьев в одно.

Либо докажите корректность описанного алгоритма поиска минимального остовного дерева, либо опровергните его, приведя контрпример, для которого алгоритм работает некорректно.

Задачи

23.1. Второе минимальное остовное дерево

Пусть $G = (V, E)$ — неориентированный связный граф с весовой функцией $w : E \to \mathbb{R}$; предположим, что $|E| \geq |V|$ и что веса всех ребер различны.

Определим второе минимальное остовное дерево следующим образом. Пусть $\mathcal{T}$ — множество всех остовных деревьев G и пусть T' — минимальное

остовное дерево G. Тогда ***вторым минимальным остовным деревом*** (second-best minimum spanning tree) будет такое остовное дерево T, что $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

а. Покажите, что минимальное остовное дерево единственное, но вторых минимальных остовных деревьев может быть несколько.

б. Пусть T — минимальное остовное дерево графа G. Докажите, что граф G содержит ребра $(u, v) \in T$ и $(x, y) \notin T$, такие, что $T - \{(u, v)\} \cup \{(x, y)\}$ является вторым минимальным остовным деревом G.

в. Пусть T — остовное дерево G, и для любых двух вершин $u, v \in V$ обозначим через $max[u, v]$ ребро максимального веса на единственном простом пути между u и v в T. Разработайте алгоритм, который за время $O(V^2)$ для заданного T вычисляет $max[u, v]$ для всех $u, v \in V$.

г. Разработайте эффективный алгоритм вычисления второго минимального остовного дерева графа G.

23.2. Минимальное остовное дерево разреженного графа

Для очень разреженного связного графа $G = (V, E)$ можно добиться дальнейшего улучшения времени работы $O(E + V \lg V)$ алгоритма Прима с использованием фибоначчиевых пирамид путем предварительной обработки G в целях уменьшения количества его вершин перед применением алгоритма Прима. В частности, для каждой вершины u мы выбираем инцидентное u ребро (u, v) с минимальным весом и помещаем это ребро в строящееся минимальное остовное дерево. Затем мы выполняем сжатие всех выбранных ребер (см. раздел Б.4). Вместо того чтобы выполнять сжатие по одному ребру, мы сначала определяем множества вершин, которые объединяются в одну новую вершину. Затем мы создаем граф, который должен был бы получиться в результате объединения этих ребер по одному, но делаем это путем "переименования" ребер соответственно множествам, в которых оказываются концы этих ребер. При этом одинаково переименованными могут оказаться одновременно несколько ребер, и в таком случае в качестве результирующего рассматривается только одно ребро с весом, равным минимальному весу среди соответствующих исходных ребер.

Изначально мы полагаем строящееся минимальное остовное дерево T пустым и для каждого ребра $(u, v) \in E$ инициализируем атрибуты $(u, v).orig = (u, v)$ и $(u, v).c = w(u, v)$. Атрибут *orig* используется для ссылки на ребро исходного графа, которое связано с данным ребром в сжатом графе. Атрибут c хранит вес ребра, и при сжатии его значение обновляется в соответствии с приведенной выше схемой выбора весов ребер. Процедура MST-REDUCE получает в качестве входных параметров G и T и возвращает сжатый граф G' с обновленными атрибутами $orig'$ и c'. Процедура также переносит ребра из G в минимальное остовное дерево T.

```
MST-REDUCE(G,T)
for каждой вершины v ∈ G.V
    v.mark = FALSE
    MAKE-SET(v)
for каждой вершины u ∈ G.V
    if u.mark == FALSE
        выбрать вершину v ∈ G.Adj[u], такую, чтобы
        значение атрибута (u,v).c было минимальным
        UNION(u,v)
        T = T ∪ {(u,v).orig}
        u.mark = v.mark = TRUE
G'.V = {FIND-SET(v) : v ∈ G.V}
G'.E = ∅
for каждого ребра (x,y) ∈ G.E
    u = FIND-SET(x)
    v = FIND-SET(y)
    if (u,v) ∉ G'.E
        G'.E = G'.E ∪ {(u,v)}
        (u,v).orig' = (x,y).orig
        (u,v).c' = (x,y).c
    else if (x,y).c < (u,v).c'
            (u,v).orig' = (x,y).orig
            (u,v).c' = (x,y).c
Построить списки смежности G'.Adj для G'
return G' и T
```

а. Пусть T — множество ребер, возвращаемое процедурой MST-REDUCE и пусть A — минимальное остовное дерево графа G', образованное вызовом MST-PRIM(G', c', r), где c' — атрибут веса для ребер из $G'.E$, а r — произвольная вершина из $G'.V$. Докажите, что $T \cup \{(x,y).orig' : (x,y) \in A\}$ представляет собой минимальное остовное дерево графа G.

б. Докажите, что $|G'.V| \leq |V|/2$.

в. Покажите, как реализовать процедуру MST-REDUCE так, чтобы время ее работы составляло $O(E)$. (*Указание:* воспользуйтесь простыми структурами данных.)

г. Предположим, что мы k раз применяем процедуру MST-REDUCE, используя полученный при очередном вызове граф G' в качестве входных данных для следующего вызова и накапливаем ребра в T. Докажите, что общее время выполнения всех k вызовов составляет $O(kE)$.

д. Предположим, что после k вызовов процедуры MST-REDUCE мы применяем алгоритм Прима, вызывая MST-PRIM(G', c', r), где G' с атрибутом веса c' получен в результате последнего вызова процедуры MST-REDUCE, а r — произвольная вершина из $G'.V$. Покажите, как выбрать k, чтобы общее время

работы составило $O(E \lg \lg V)$. Докажите, что ваш выбор k минимизирует общее асимптотическое время работы.

е. Для каких значений $|E|$ (выраженных через $|V|$) алгоритм Прима с предварительным сжатием эффективнее алгоритма Прима без сжатия?

23.3. Узкое остовное дерево

Назовем ***узким остовным деревом*** (bottleneck spanning tree) T неориентированного графа G остовное дерево G, в котором наибольший вес ребра минимален среди всех возможных остовных деревьев, и назовем этот вес значением узкого остовного дерева.

а. Докажите, что минимальное остовное дерево является узким остовным деревом.

Из части (а) следует, что поиск узкого остовного дерева не сложнее поиска минимального остовного дерева. В оставшейся части задачи мы покажем, что его можно найти за линейное время.

б. Разработайте алгоритм, который для данного графа G и целого числа b за линейное время определяет, превышает ли значение узкого остовного дерева число b.

в. Воспользуйтесь своим алгоритмом из п. (б) как подпрограммой в алгоритме решения задачи поиска узкого остовного дерева за линейное время. (*Указание:* можно воспользоваться подпрограммой сжатия множеств ребер, как в процедуре MST-REDUCE, описанной в задаче 23.2.)

23.4. Альтернативные алгоритмы поиска минимальных остовных деревьев

В этой задаче мы приведем псевдокоды трех различных алгоритмов. Каждый из них принимает в качестве входных данных граф и возвращает множество ребер T. Для каждого из алгоритмов требуется доказать, что T является минимальным остовным деревом графа, или доказать, что это не так. Кроме того, опишите эффективную реализацию каждого из алгоритмов, независимо от того, вычисляет он минимальное остовное дерево или нет.

а. MAYBE-MST-A(G, w)

```
Отсортировать ребра в невозрастающем порядке их весов w
T = E
for каждого ребра e в отсортированном порядке
    if T − {e} является связным графом
        T = T − {e}
return T
```

б. MAYBE-MST-B(G, w)

```
T = ∅
for каждого ребра e, взятого в произвольном порядке
    if T ∪ {e} не имеет циклов
        T = T ∪ {e}
return T
```

в. MAYBE-MST-C(G, w)

```
T = ∅
for каждого ребра e, взятого в произвольном порядке
    T = T ∪ {e}
    if T имеет цикл c
        Пусть e′ — ребро максимального веса в c
        T = T − {e′}
return T
```

Заключительные замечания

В книге Таржана (Tarjan) [328] содержится превосходный обзор задач, связанных с поиском минимальных остовных деревьев, и дополнительную информацию о них. История данной задачи изложена Грехемом (Graham) и Хеллом (Hell) [150].

Таржан указывает, что впервые алгоритм для поиска минимальных остовных деревьев был описан в 1926 году в статье О. Борувки (O. Borůvka). Его алгоритм состоит в выполнении $O(\lg V)$ итераций процедуры MST-REDUCE, описанной в задаче 23.2. Алгоритм Крускала описан в [221] в 1956 году, а алгоритм, известный как алгоритм Прима, — в работе Прима (Prim) [283], хотя до этого он был открыт В. Ярником (V. Jarník) в 1930 году.

Причина, по которой жадные алгоритмы эффективно решают задачу поиска минимальных остовных деревьев, заключается в том, что множество лесов графа образует матроид (см. раздел 16.4).

Когда $|E| = \Omega(V \lg V)$, алгоритм Прима, реализованный с использованием фибоначчиевых пирамид, имеет время работы $O(E)$. Для более разреженных графов использование комбинации идей из алгоритма Прима, алгоритмов Крускала и Борувки вместе с применением сложных структур данных дало возможность Фредману (Fredman) и Таржану [113] разработать алгоритм, время работы которого равно $O(E \lg^* V)$. Габов (Gabow), Галил (Galil), Спенсер (Spencer) и Таржан [119] усовершенствовали этот алгоритм, доведя время его работы до $O(E \widehat{\alpha}(E, V))$. Чазел (Chazelle) [59] разработал алгоритм со временем работы $O(E\ \widehat{\alpha}(E, V))$, где $\widehat{\alpha}(E, V)$ — функция, обратная функции Аккермана (см. главу 21). В отличие от перечисленных ранее, алгоритм Чазела не является жадным.

С задачей поиска минимального остовного дерева связана задача ***проверки остовного дерева*** (spanning tree verification), в которой для данного графа $G =$

(V, E) и дерева $T \subseteq E$ требуется определить, является ли T минимальным остовным деревом G. Кинг (King) [202] разработал алгоритм решения данной задачи за линейное время, основанный на работах Комлёса (Komlós) [214] и Диксона (Dixon), Рауха (Rauch) и Таржана [89].

Все описанные выше алгоритмы — детерминированные и относятся к модели на основе сравнений, описанной в главе 8. Каргер (Karger), Клейн (Klein) и Таржан [194] разработали рандомизированный алгоритм поиска минимальных остовных деревьев, математическое ожидание времени работы которого составляет $O(V + E)$. Этот алгоритм использует рекурсию наподобие алгоритма с линейным временем работы из раздела 9.3: рекурсивный вызов для вспомогательной задачи определяет подмножество ребер E', которое не может находиться ни в одном минимальном остовном дереве. Другой рекурсивный вызов, работающий с подмножеством $E - E'$, строит минимальное остовное дерево. Алгоритм также использует идеи из алгоритмов Борувки и Кинга.

Фредман и Виллард (Willard) [115] показали, как найти минимальное остовное дерево за время $O(V + E)$ с использованием детерминированного алгоритма, не основанного на сравнениях. Их алгоритм предполагает, что данные представляют собой b-битовые целые числа и что память компьютера состоит из адресуемых b-битовых слов.

Глава 24. Кратчайшие пути из одной вершины

Водителю автомобиля нужно найти самый короткий путь из Киева в Запорожье. Допустим, у него есть карта Украины, на которой указаны расстояния между каждой парой пересечений дорог. Как найти кратчайший маршрут?

Один из возможных способов — пронумеровать все маршруты из Киева в Запорожье, просуммировать длины участков на каждом маршруте и выбрать кратчайший из них. Однако легко понять, что даже если исключить маршруты, содержащие циклы, получится очень много вариантов, большинство которых просто не имеет смысла рассматривать. Например, очевидно, что маршрут из Киева в Запорожье через Львов — далеко не лучший выбор. Точнее говоря, такой маршрут никуда не годится, потому что Львов находится относительно Киева совсем в другой стороне.

В этой главе и в главе 25 будет показано, как эффективно решаются такие задачи. ***В задаче о кратчайшем пути*** (shortest-paths problem) задается взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$, отображающей ребра на их веса, значения которых выражаются действительными числами. ***Вес*** (weight) пути $p = \langle v_0, v_1, \ldots, v_k \rangle$ равен суммарному весу входящих в него ребер:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Вес кратчайшего пути (shortest-path weight) $\delta(u, v)$ из вершины u в вершину v определяется соотношением

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} , & \text{если существует путь из } u \text{ в } v , \\ \infty & \text{в противном случае} . \end{cases}$$

Тогда по определению ***кратчайший путь*** (shortest path) из вершины u в вершину v — это любой путь, вес которого удовлетворяет соотношению $w(p) = \delta(u, v)$.

В примере, в котором рассматривается маршрут из Киева в Запорожье, карту дорог можно смоделировать в виде графа, вершины которого представляют перекрестки дорог, а ребра — отрезки дорог между перекрестками, причем вес каждого ребра равен расстоянию между соответствующими перекрестками. Цель — найти кратчайший путь от заданного перекрестка в Киеве (например, между ули-

цами Клавдиевской и Корсуньской) к заданному перекрестку в Запорожье (скажем, между улицами Панфиловцев и Патриотической).

Вес каждого из ребер можно интерпретировать не как расстояние, а как другую метрику. Часто они используются для представления временны́х интервалов, стоимости, штрафов, убытков или любой другой величины, которая линейно накапливается по мере продвижения вдоль ребер графа и которую нужно свести к минимуму.

Алгоритм поиска в ширину, описанный в разделе 22.2, представляет собой алгоритм поиска кратчайшего пути в невзвешенном графе, т.е. в графе, каждому ребру которого приписывается единичный вес. Поскольку многие концепции, применяемые в алгоритме поиска в ширину, возникают при исследовании задачи о кратчайшем пути по взвешенным графам, рекомендуется перед дальнейшим чтением освежить в памяти материал раздела 22.2.

Варианты

Настоящая глава посвящена ***задаче о кратчайших путях из одной вершины*** (single-source shortest-paths problem), в которой для заданного графа $G = (V, E)$ требуется найти кратчайшие пути, которые начинаются в определенной ***исходной вершине*** (source vertex) $s \in V$ (для краткости будем именовать ее истоком) и заканчиваются в каждой из вершин $v \in V$. Предназначенный для решения этой задачи алгоритм позволяет решать многие другие задачи, в том числе перечисленные ниже.

Задача о кратчайших путях в одну вершину. Требуется найти кратчайшие пути в заданную ***целевую вершину*** (destination vertex) t, которые начинаются в каждой из вершин v. Поменяв направление каждого принадлежащего графу ребра, эту задачу можно свести к задаче о единой исходной вершине.

Задача о кратчайшем пути между заданной парой вершин. Требуется найти кратчайший путь из заданной вершины u в заданную вершину v. Если решена задача поиска кратчайших путей из заданной исходной вершины u, то эта задача также решается. Более того, все известные для решения данной задачи алгоритмы имеют то же время работы в наихудшем случае, что и наилучшие алгоритмы поиска кратчайших путей из одной вершины.

Задача о кратчайшем пути между всеми вершинами. Требуется найти кратчайший путь из каждой вершины u в каждую вершину v. Эту задачу также можно решить с помощью алгоритма, предназначенного для решения задачи об одной исходной вершине, однако обычно она решается быстрее. Кроме того, структура этой задачи представляет интерес сама по себе. В главе 25 задача обо всех парах вершин исследуется более подробно.

Оптимальная подструктура кратчайших путей

Алгоритмы поиска кратчайших путей обычно основаны на том свойстве, что кратчайший путь между двумя вершинами содержит в себе другие кратчайшие пути. (В основе описанного в главе 26 алгоритма Эдмондса–Карпа (Edmonds–

Карр), предназначенного для поиска максимального потока, также лежит это свойство.) Вспомним, что свойство оптимальной подструктуры — один из ключевых индикаторов применимости и динамического программирования (глава 15), и жадного метода (глава 16). Алгоритм Дейкстры (Dijkstra), с которым мы ознакомимся в разделе 24.3, представляет собой жадный алгоритм, а алгоритм Флойда–Уоршелла (Floyd–Warshall), предназначенный для поиска кратчайшего пути между всеми парами вершин (см. раздел 25.2), — алгоритм динамического программирования. В сформулированной ниже лемме данное свойство оптимальной структуры определяется более точно.

Лемма 24.1 (Подпути кратчайших путей есть кратчайшие пути)

Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$ — кратчайший путь из вершины v_0 в вершину v_k в заданном взвешенном ориентированном графе $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ и пусть для любых i и j, таких, что $0 \leq i \leq j \leq k$, путь $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ является подпутем p из вершины v_i в вершину v_j. Тогда p_{ij} является кратчайшим путем из v_i в v_j.

Доказательство. Если разложить путь p на составные части $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, то будет выполняться соотношение $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Теперь предположим, что существует путь p'_{ij} из вершины v_i в вершину v_j с весом $w(p'_{ij}) < w(p_{ij})$. Тогда $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ представляет собой путь из v_0 в v_k, вес которого $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ меньше, чем $w(p)$, что противоречит предположению о том, что p является кратчайшим путем из вершины v_0 в вершину v_k. ■

Ребра с отрицательным весом

В некоторых экземплярах задачи о кратчайшем пути из фиксированного истока веса ребер могут принимать отрицательные значения. Если граф $G = (V, E)$ не содержит циклов с отрицательным весом, достижимых из истока s, то вес кратчайшего пути $\delta(s, v)$ остается вполне определенной величиной для каждой вершины $v \in V$, даже если он принимает отрицательное значение. Если же такой цикл достижим из истока s, веса кратчайших путей перестают быть вполне определенными величинами. В этой ситуации ни один путь из истока s в любую из вершин цикла не может быть кратчайшим, потому что всегда можно найти путь с меньшим весом, который проходит по предложенному "кратчайшему" пути, а затем обходит цикл с отрицательным весом. Если на некотором пути из вершины s к вершине v встречается цикл с отрицательным весом, мы определяем $\delta(s, v) = -\infty$.

На рис. 24.1 проиллюстрировано влияние наличия отрицательных весов и циклов с отрицательным весом на веса кратчайших путей. Поскольку из вершины s в вершину a ведет всего один путь (путь $\langle s, a \rangle$), то выполняется равенство $\delta(s, a) = w(s, a) = 3$. Аналогично имеется всего один путь из вершины s в вершину b, поэтому $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Из вершины s в вершину c можно провести бесконечно большое количество путей: $\langle s, c \rangle$,

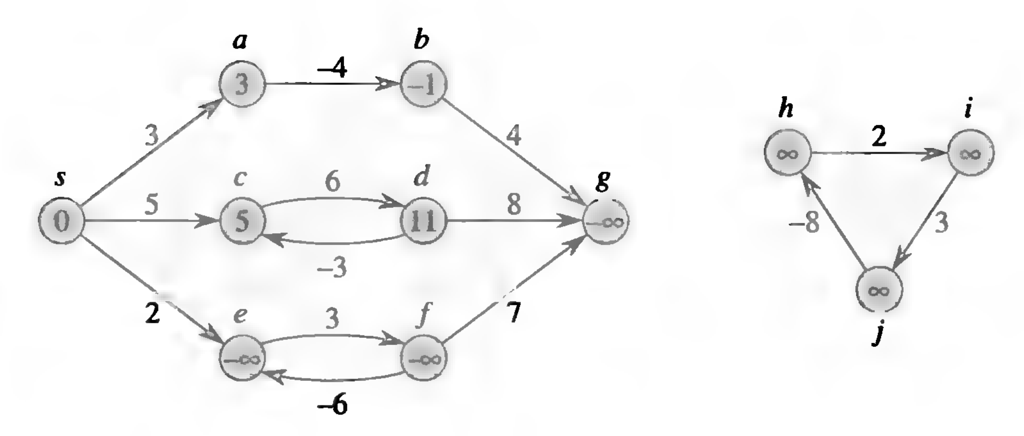


Рис. 24.1. Ребра с отрицательными весами в ориентированном графе. Для каждой вершины указан вес кратчайшего пути до нее из вершины s. Поскольку вершины e и f образуют цикл с отрицательным весом, достижимый из s, веса соответствующих кратчайших путей равны $-\infty$. Вершина g достижима из вершины, вес кратчайшего пути к которой равен $-\infty$, поэтому она также имеет вес кратчайшего пути, равный $-\infty$. Вершины, такие как h, i и j, недостижимы из s, поэтому для каждой из них вес кратчайшего пути равен ∞, несмотря на то что они находятся в цикле с отрицательным весом.

$\langle s,c,d,c \rangle$, $\langle s,c,d,c,d,c \rangle$ и т.д. Поскольку вес цикла $\langle c,d,c \rangle$ равен $6+(-3)=3>0$, кратчайшим путем из вершины s в вершину c является путь $\langle s, c \rangle$, вес которого равен $\delta(s,c) = w(s,c) = 5$. Аналогично кратчайшим путем из вершины s в вершину d является путь $\langle s, c, d \rangle$ с весом $\delta(s,d) = w(s,c) + w(c,d) = 11$. Точно так же имеется бесконечно большое количество путей из s в вершину e: $\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$ и т.д. Однако, поскольку вес цикла $\langle e, f, e \rangle$ равен $3 + (-6) = -3 < 0$, т.е. он отрицательный, не существует кратчайшего пути из вершины s в вершину e. Обходя цикл с отрицательным весом $\langle e, f, e \rangle$ сколько угодно раз, можно построить путь из s в вершину e, вес которого будет выражаться каким угодно большим по модулю отрицательным числом, поэтому $\delta(s,e) = -\infty$ и аналогично $\delta(s,f) = -\infty$. Поскольку вершина g достижима из вершины f, можно также найти пути из вершины s в вершину g с произвольно большими отрицательными весами, а потому $\delta(s,g) = -\infty$. Вершины h, i и j также образуют цикл с отрицательным весом. Однако они недостижимы из вершины s, поэтому $\delta(s,h) = \delta(s,i) = \delta(s,j) = \infty$.

В некоторых алгоритмах поиска кратчайшего пути, таких как алгоритм Дейкстры, предполагается, что вес любого из ребер входного графа неотрицательный, как это было в примере с дорожной картой. В других алгоритмах, таких как алгоритм Беллмана–Форда (Bellman–Ford), ребра входных графов могут иметь отрицательный вес. Эти алгоритмы дают корректный ответ, если циклы с отрицательным весом недостижимы из истока. Обычно, если такой цикл с отрицательным весом существует, подобный алгоритм способен выявить его наличие и сообщить об этом.

Циклы

Может ли кратчайший путь содержать цикл? Только что мы убедились в том, что он не может содержать цикл с отрицательным весом. В него также не может входить цикл с положительным весом, поскольку в результате удаления этого

цикла из пути получится путь, который исходит из того же истока и заканчивается в той же вершине, но обладает меньшим весом. То есть, если $p = \langle v_0, v_1, \ldots, v_k \rangle$ является путем, а $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ — цикл с положительным весом на этом пути (так что $v_i = v_j$ и $w(c) > 0$), то путь $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ имеет вес $w(p') = w(p) - w(c) < w(p)$, а значит, p не может быть кратчайшим путем из v_0 в v_k.

Остаются только циклы с нулевым весом. Однако из пути можно удалить цикл с нулевым весом, в результате чего получится другой путь с тем же весом. Таким образом, если существует кратчайший путь из истока s в целевую вершину v, содержащий цикл с нулевым весом, то существует и другой кратчайший путь из истока s в целевую вершину v, в котором этот цикл не содержится. Если кратчайший путь содержит циклы с нулевым весом, эти циклы можно поочередно удалять до тех пор, пока не получится кратчайший путь, в котором циклы отсутствуют. Поэтому без потери общности можно предположить, что когда мы находим кратчайшие пути, они не содержат циклов. Поскольку в любой ациклический путь в графе $G = (V, E)$ входит не более $|V|$ различных вершин, в нем содержится не более $|V| - 1$ ребер. Таким образом, можно ограничиться рассмотрением кратчайших путей, состоящих не более чем из $|V| - 1$ ребер.

Представление кратчайших путей

Часто требуется вычислить не только вес каждого из кратчайших путей, но и входящие в их состав вершины. Представление, используемое для кратчайших путей, аналогично тому, которое используется для описанных в разделе 22.2 деревьев поиска в ширину. В заданном графе $G = (V, E)$ для каждой вершины $v \in V$ поддерживается атрибут ***предшественник*** (predecessor) $v.\pi$, в роли которого выступает либо другая вершина, либо значение NIL. В рассмотренных в этой главе алгоритмах поиска кратчайших путей атрибуты π присваиваются таким образом, что цепочка предшественников, которая начинается в вершине v, позволяет проследить путь, обратный кратчайшему пути из вершины s в вершину v. Таким образом, для заданной вершины v, у которой $v.\pi \neq \text{NIL}$, с помощью описанной в разделе 22.2 процедуры PRINT-PATH(G, s, v) можно вывести кратчайший путь из вершины s в вершину v.

Однако до тех пор, пока алгоритм поиска кратчайших путей не закончил свою работу, значения π не обязательно указывают кратчайшие пути. Как и при поиске в ширину, нас будет интересовать ***подграф предшествования*** (predecessor subgraph) $G_\pi = (V_\pi, E_\pi)$, порожденный значениями π. Как и ранее, определим множество вершин V_π как множество, состоящее из тех вершин графа G, предшественниками которых не являются значения NIL, а также включающее исток s:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\} .$$

Множество ориентированных ребер E_π — это множество ребер, порожденных значениями π у вершин из множества V_π:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} .$$

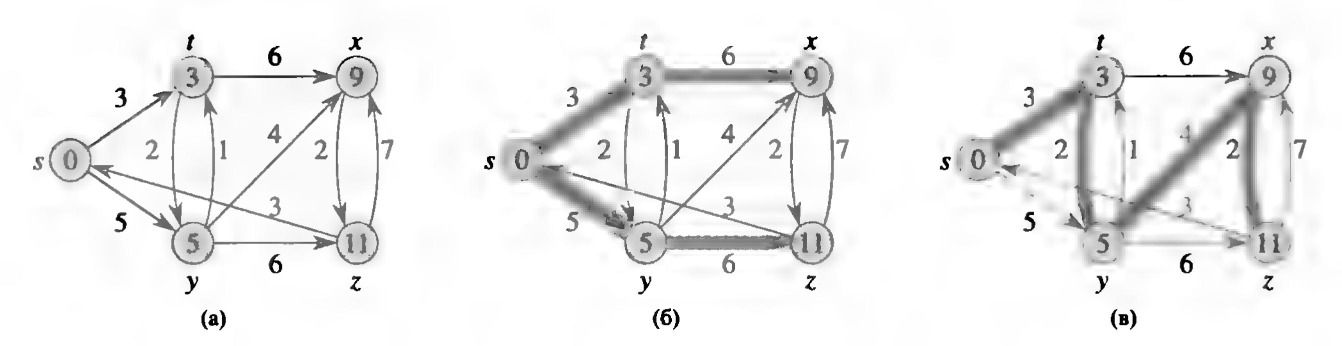


Рис. 24.2. **(а)** Взвешенный ориентированный граф с весами кратчайших путей из истока s. **(б)** Заштрихованные ребра образуют дерево кратчайших путей с корнем в истоке s. **(в)** Еще одно дерево кратчайших путей с тем же корнем.

Далее будет доказано, что значения π, полученные с помощью описанных в этой главе алгоритмов, обладают тем свойством, что после завершения этих алгоритмов G_π является "деревом кратчайших путей". Неформально это дерево можно описать как корневое дерево, содержащее кратчайший путь из истока s к каждой вершине, достижимой из вершины s. Оно похоже на дерево поиска в ширину, знакомое нам из раздела 22.2, но содержит кратчайшие пути из истока, определенные не с помощью количества ребер, а с помощью значений их весов. Дадим более точное определение. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Предположим, что в нем не содержится циклов с отрицательным весом, достижимых из истока $s \in V$, так что кратчайшие пути вполне определены. Тогда ***дерево кратчайших путей*** (shortest-paths tree) с корнем в вершине s представляет собой ориентированный подграф $G' = (V', E')$, в котором множества $V' \subseteq V$ и $E' \subseteq E$ определяются следующими условиями.

1. Множество V' представляет собой множество вершин, достижимых из истока s графа G.
2. Граф G' образует корневое дерево с корнем s.
3. Для всех $v \in V'$ однозначно определяемый простой путь из вершины s в вершину v в графе G' является кратчайшим путем из s в v в G.

Кратчайшие пути, как и деревья кратчайших путей, не обязательно единственны. Например, на рис. 24.2 изображен взвешенный ориентированный граф, на котором обозначен вес каждого из кратчайших путей из истока s, а также два дерева кратчайших путей с одним и тем же корнем.

Ослабление

В описанных в этой главе алгоритмах используется метод ***релаксации***, или ***ослабления*** (relaxation). Для каждой вершины $v \in V$ поддерживается атрибут $v.d$, представляющий собой верхнюю границу веса, которым обладает кратчайший путь из истока s в вершину v. Мы называем атрибут $v.d$ ***оценкой кратчайшего пути*** (shortest-path estimate). Инициализация оценок кратчайших путей и предшественников проводится в приведенной ниже процедуре, время работы которой равно $\Theta(V)$.

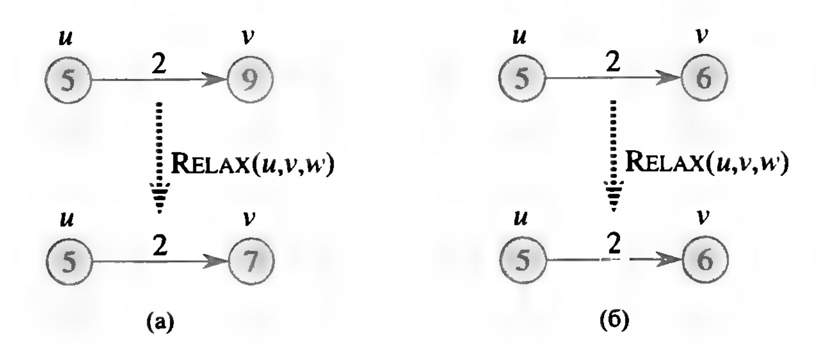


Рис. 24.3. Ослабление ребра (u, v) с весом $w(u, v) = 2$. Для каждой вершины приведена оценка ее кратчайшего пути. **(а)** Поскольку перед ослаблением $v.d > u.d + w(u, v)$, значение $v.d$ уменьшается. **(б)** Здесь перед ослаблением ребра $v.d \le u.d + w(u, v)$, так что ослабление оставляет значение $v.d$ неизменным.

```
INITIALIZE-SINGLE-SOURCE(G, s)
for каждой вершины v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

После инициализации для всех $v \in V$ выполняется $v.\pi = \text{NIL}$, $s.d = 0$ и $v.d = \infty$ для $v \in V - \{s\}$.

Процесс ***ослабления*** (relaxation) ребра (u, v) заключается в проверке, нельзя ли улучшить найденный к этому моменту кратчайший путь к вершине v, проведя его через вершину u, а также в обновлении атрибутов $v.d$ и $v.\pi$ при наличии такой возможности. Ослабление[1] может уменьшить оценку кратчайшего пути $v.d$ и обновить атрибут $v.\pi$ вершины v. Приведенный ниже код выполняет ослабление ребра (u, v) за время $O(1)$.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

На рис. 24.3 приведены два примера ослабления ребра; в одном из них оценка кратчайшего пути понижается, а в другом — не происходит никаких изменений, связанных с оценками.

В каждом из описанных в этой главе алгоритмов сначала вызывается процедура INITIALIZE-SINGLE-SOURCE, а затем выполняется ослабление ребер. Более того, ослабление — единственная операция, изменяющая оценки кратчайших путей и предшественников. Описанные в этой главе алгоритмы различаются тем, сколь-

[1]Может показаться странным, что термин "ослабление" используется для операции, которая уточняет верхнюю границу. Этот термин определился исторически. Результат этапа ослабления можно рассматривать как ослабление ограничения $v.d \le u.d + w(u, v)$, которое должно выполняться согласно неравенству треугольника (лемма 24.10), если $u.d = \delta(s, u)$ и $v.d = \delta(s, v)$. Другими словами, если справедливо неравенство $v.d \le u.d + w(u, v)$, оно выполняется без "давления", поэтому данное условие "ослабляется".

ко раз в них проводится ослабление ребер, а также порядком ребер, над которыми выполняется ослабление. В алгоритме Дейкстры и алгоритме поиска кратчайших путей в ориентированных ациклических графах каждое ребро ослабляется ровно по одному разу. В алгоритме Беллмана–Форда (Bellman–Ford) каждое ребро ослабляется $|V| - 1$ раз.

Свойства кратчайших путей и ослабление

Для доказательства корректности описанных в этой главе алгоритмов будут использоваться некоторые свойства кратчайших путей и ослабления. Здесь эти свойства лишь сформулированы, формально они будут доказаны в разделе 24.5. Чтобы не нарушалась целостность изложения, каждое приведенное здесь свойство содержит номер соответствующей ему леммы или следствия из раздела 24.5. В последних пяти свойствах, которые относятся к оценкам кратчайших путей или подграфу предшествования, неявно подразумевается, что граф инициализирован с помощью вызова процедуры INITIALIZE-SINGLE-SOURCE(G, s) и что единственный способ изменения оценок кратчайших путей и подграфа предшествования — выполнение некоторой последовательности этапов ослабления.

Неравенство треугольника (лемма 24.10)

Для каждого ребра $(u, v) \in E$ выполняется неравенство $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Свойство верхней границы (лемма 24.11)

Для всех вершин $v \in V$ всегда выполняется неравенство $v.d \ge \delta(s, v)$, а после того как величина $v.d$ достигает значения $\delta(s, v)$, она больше не изменяется.

Свойство отсутствия пути (следствие 24.12)

Если из вершины s в вершину v нет пути, то всегда выполняется соотношение $v.d = \delta(s, v) = \infty$.

Свойство сходимости (лемма 24.14)

Если $s \leadsto u \to v$ является кратчайшим путем в G для некоторых $u, v \in V$ и если $u.d = \delta(s, u)$ в любой момент до ослабления ребра (u, v), то $v.d = \delta(s, v)$ в любой момент после этого.

Свойство ослабления пути (лемма 24.15)

Если $p = \langle v_0, v_1, \ldots, v_k \rangle$ является кратчайшим путем из $s = v_0$ в v_k и если мы ослабляем ребра p в порядке $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, то $v_k.d = \delta(s, v_k)$. Это свойство выполняется независимо от других этапов ослабления, даже если они чередуются с ослаблением ребер, принадлежащих пути p.

Свойство подграфа предшествования (лемма 24.17)

Если для всех вершин $v \in V$ выполняется $v.d = \delta(s, v)$, то подграф предшествования представляет собой дерево кратчайших путей с корнем в истоке s.

Краткое содержание главы

В разделе 24.1 представлен алгоритм Беллмана–Форда, позволяющий решить задачу о кратчайшем пути из фиксированного истока в общем случае, когда вес

любого ребра может быть отрицательным. Этот алгоритм отличается своей простотой. К его достоинствам также относится то, что он определяет, содержится ли в графе цикл с отрицательным весом, достижимый из истока. В разделе 24.2 приводится алгоритм с линейным временем выполнения, предназначенный для построения кратчайших путей из одной вершины в ориентированном ациклическом графе. В разделе 24.3 рассматривается алгоритм Дейкстры, который характеризуется меньшим временем выполнения, чем алгоритм Беллмана–Форда, но требует, чтобы вес каждого из ребер был неотрицательным. В разделе 24.4 показано, как с помощью алгоритма Беллмана–Форда можно решить частный случай задачи линейного программирования. Наконец, в разделе 24.5 доказываются сформулированные выше свойства кратчайших путей и ослабления.

Примем некоторые соглашения, необходимые для выполнения арифметических операций с бесконечно большими величинами. Будем считать, что для любого действительного числа $a \neq -\infty$ выполняется соотношение $a+\infty = \infty+a = \infty$. Кроме того, чтобы наши доказательства сохраняли силу при наличии циклов с отрицательным весом, будем считать, что для любого действительного числа $a \neq \infty$ выполняется соотношение $a + (-\infty) = (-\infty) + a = -\infty$.

Во всех описанных в этой главе алгоритмах предполагается, что ориентированный граф G хранится в виде списков смежных вершин. Кроме того, вместе с каждым ребром хранится его вес, так что при просмотре каждого списка смежности вес каждого из его ребер можно определить за время $O(1)$.

24.1. Алгоритм Беллмана–Форда

Алгоритм Беллмана–Форда (Bellman–Ford algorithm) решает задачу о кратчайшем пути из одной вершины в общем случае, когда вес каждого из ребер может быть отрицательным. Для заданного взвешенного ориентированного графа $G = (V, E)$ с истоком s и весовой функцией $w : E \to \mathbb{R}$ алгоритм Беллмана–Форда возвращает логическое значение, указывающее, содержится ли в графе цикл с отрицательным весом, достижимый из истока. Если такой цикл существует, алгоритм указывает, что решения не существует. Если же таких циклов нет, алгоритм выдает кратчайшие пути и их веса.

В этом алгоритме используется ослабление, в результате которого величина $v.d$, представляющая собой оценку веса кратчайшего пути из истока s к каждой из вершин $v \in V$, постепенно уменьшается до тех пор, пока не станет равной фактическому весу кратчайшего пути $\delta(s, v)$. Значение TRUE возвращается алгоритмом тогда и только тогда, когда граф не содержит циклов с отрицательным весом, достижимых из истока.

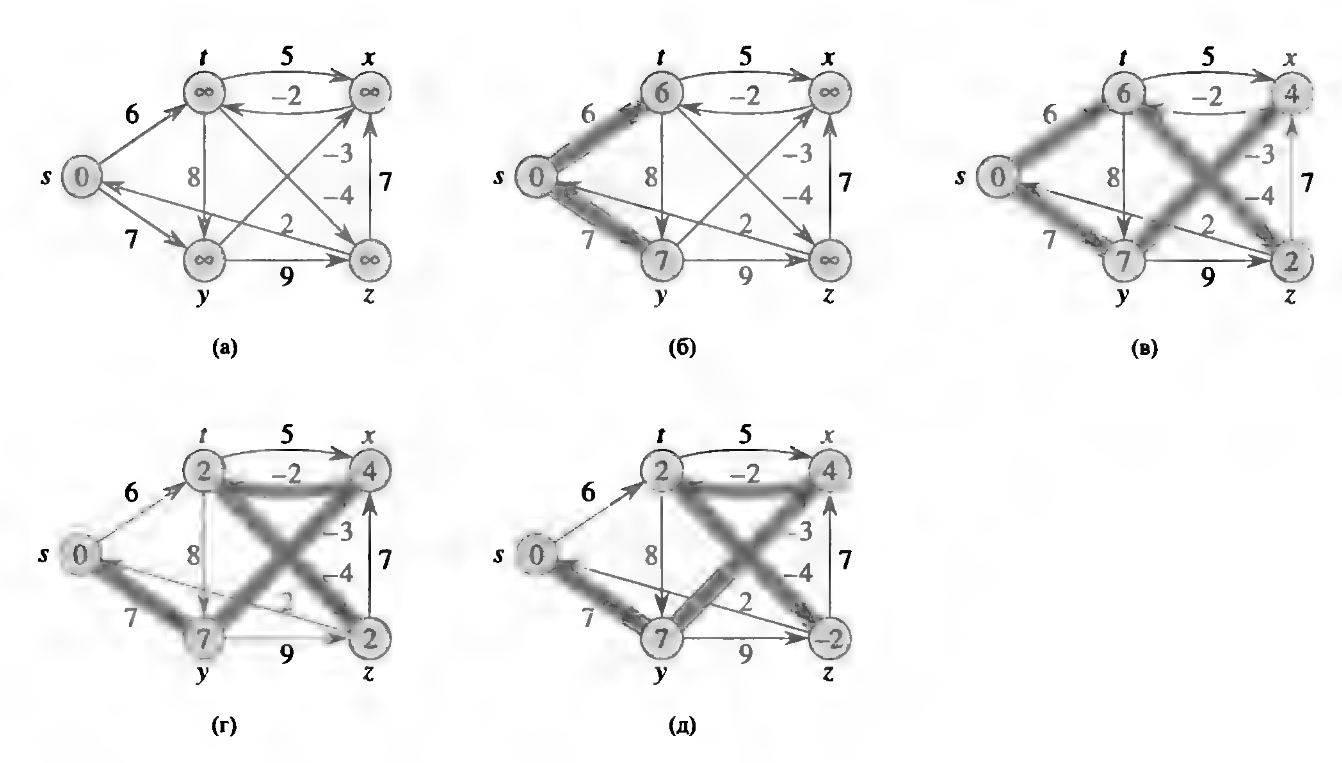


Рис. 24.4. Выполнение алгоритма Беллмана–Форда. Истоком является вершина s. В вершинах показаны значения d, а заштрихованные ребра указывают предшественников: если ребро (u, v) заштриховано, то $v.\pi = u$. В этом конкретном примере каждый проход ослабляет ребра в порядке $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(а)** Ситуация непосредственно перед первым проходом по ребрам. **(б)–(д)** Ситуация после каждого последующего прохода по ребрам. Значения d и π в части (д) являются окончательными значениями. Алгоритм Беллмана–Форда в данном примере возвращает значение TRUE.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for каждого ребра (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for каждого ребра (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

На рис. 24.4 проиллюстрирована работа алгоритма BELLMAN-FORD с графом, содержащим 5 вершин. После инициализации значений d и π у всех вершин в строке 1 алгоритм выполняет $|V| - 1$ проходов по ребрам графа. Каждый проход представляет собой одну итерацию цикла **for** в строках 2–4 и состоит из однократного ослабления каждого ребра графа. На рис. 24.4, (б)–(д) показано состояние алгоритма после каждого из четырех проходов по ребрам. После выполнения $|V| - 1$ проходов в строках 5–8 выполняется проверка наличия цикла с отрицательным весом, и возвращается соответствующее логическое значение. (Чуть позже вы поймете, почему работает эта проверка.)

Время работы алгоритма Беллмана–Форда составляет $O(VE)$, поскольку инициализация в строке 1 занимает время $\Theta(V)$, на каждый из $|V| - 1$ проходов по

ребрам в строках 2–4 требуется время $\Theta(E)$, а на выполнение цикла **for** в строках 5–7 затрачивается время $O(E)$.

Чтобы доказать корректность алгоритма Беллмана–Форда, сначала покажем, что при отсутствии циклов с отрицательным весом он правильно вычисляет веса кратчайших путей для всех вершин, достижимых из истока.

Лемма 24.2

Пусть $G = (V, E)$ является взвешенным ориентированным графом с истоком s и весовой функцией $w : E \to \mathbb{R}$, который не содержит циклов с отрицательным весом, достижимых из вершины s. Тогда по завершении $|V| - 1$ итераций цикла **for** в строках 2–4 алгоритма BELLMAN-FORD для всех вершин v, достижимых из вершины s, выполняется равенство $v.d = \delta(s, v)$.

Доказательство. Докажем сформулированную лемму, воспользовавшись свойством ослабления пути. Рассмотрим произвольную вершину v, достижимую из вершины s. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$ и $v_k = v$, представляет собой кратчайший ациклический путь из вершины s в вершину v. Поскольку кратчайший путь является простым, путь p содержит не более $|V| - 1$ ребер, так что $k \leq |V| - 1$. При каждой из $|V| - 1$ итераций цикла **for** в строках 2–4 ослабляются все $|E|$ ребер. Среди ребер, ослабленных во время i-й итерации ($i = 1, 2, \ldots, k$), находится ребро (v_{i-1}, v_i). Таким образом, согласно свойству ослабления путей, выполняется цепочка равенств $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

Следствие 24.3

Пусть $G = (V, E)$ является взвешенным ориентированным графом с истоком s и весовой функцией $w : E \to \mathbb{R}$ и пусть граф G не содержит достижимых из s циклов с отрицательным весом. Тогда для каждой вершины $v \in V$ путь из вершины s в вершину v существует тогда и только тогда, когда после применения к графу G процедуры BELLMAN-FORD выполняется неравенство $v.d < \infty$.

Доказательство. Доказательство этого следствия остается читателю в качестве упр. 24.1.2. ■

Теорема 24.4 (Корректность алгоритма Беллмана–Форда)

Пусть алгоритм BELLMAN-FORD применяется к взвешенному ориентированному графу $G = (V, E)$ с истоком s и весовой функцией $w : E \to \mathbb{R}$. Если граф G не содержит циклов с отрицательным весом, достижимых из вершины s, то этот алгоритм возвращает значение TRUE, для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$ и подграф предшествования G_π является деревом кратчайших путей с корнем в вершине s. Если же граф G содержит цикл с отрицательным весом, достижимый из вершины s, то алгоритм возвращает значение FALSE.

Доказательство. Предположим, что граф G не содержит циклов с отрицательным весом, достижимых из истока s. Сначала докажем, что по завершении работы алгоритма для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$. Если вершина v достижима из истока s, то доказательством этого утверждения служит

лемма 24.2. Если же вершина v не достижима из вершины s, то это утверждение следует из свойства отсутствия пути. Таким образом, данное утверждение доказано. Из свойства подграфа предшествования и этого утверждения следует, что граф G_π — дерево кратчайших путей. А теперь с помощью обоснованного выше утверждения покажем, что алгоритм BELLMAN-FORD возвращает значение TRUE. По завершении работы алгоритма для всех ребер $(u, v) \in E$ выполняется соотношение

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \qquad \text{(согласно неравенству треугольника)} \\ &= u.d + w(u, v) \ , \end{aligned}$$

так что ни одна из проверок, выполненных в строке 6, не приведет к тому, что алгоритм BELLMAN-FORD возвратит значение FALSE. Следовательно, ему ничего не остается, как возвратить значение TRUE.

Теперь предположим, что граф G содержит цикл с отрицательным весом, достижимый из истока s; пусть это будет цикл $c = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = v_k$. Тогда

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \ . \qquad (24.1)$$

Чтобы воспользоваться методом "от противного", предположим, что алгоритм BELLMAN-FORD возвращает значение TRUE. Тогда для значений $i = 1, 2, \ldots, k$ выполняются неравенства $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$. Просуммировав их по всему циклу c, получим

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \ . \end{aligned}$$

Поскольку $v_0 = v_k$, каждая вершина в цикле c в каждой сумме $\sum_{i=1}^{k} v_i.d$ и $\sum_{i=1}^{k} v_{i-1}.d$ появляется ровно по одному разу, так что

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \ .$$

Кроме того, согласно следствию 24.3 атрибут $v_i.d$ при $i = 1, 2, \ldots, k$ принимает конечные значения. Таким образом, справедливо неравенство

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \ ,$$

что противоречит неравенству (24.1). Итак, мы приходим к выводу, что алгоритм BELLMAN-FORD возвращает значение TRUE, если граф G не содержит циклов с отрицательным весом, достижимых из истока, и значение FALSE — в противном случае. ■

Упражнения

24.1.1

Примените алгоритм BELLMAN-FORD к ориентированному графу, показанному на рис. 24.4, в котором в качестве истока используется вершина z. В процессе каждого прохода ослабление ребер должно выполняться в том же порядке, что и на рисунке. Составьте список значений, которые принимают атрибуты d и π после каждого прохода. А теперь измените вес ребра (z, x), присвоив ему значение 4, и выполните алгоритм снова, используя в качестве истока вершину s.

24.1.2

Докажите следствие 24.3.

24.1.3

Пусть дан взвешенный ориентированный граф $G = (V, E)$, который не содержит циклов с отрицательным весом. Для каждой пары вершин $u, v \in V$ найдем минимальное количество ребер в кратчайшем пути от v к u (длина пути определяется его весом, а не количеством ребер). Пусть m — максимальное из всех полученных таким образом количеств ребер. Предложите простое изменение алгоритма BELLMAN-FORD, позволяющее ему завершаться после выполнения $m+1$ проходов, даже если m заранее неизвестно.

24.1.4

Модифицируйте алгоритм BELLMAN-FORD таким образом, чтобы в нем для всех вершин v, для которых на одном из путей от истока к v имеется цикл с отрицательным весом, атрибутам $v.d$ присваивались значения $-\infty$.

24.1.5 ★

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Разработайте алгоритм со временем работы $O(VE)$, который позволял бы для каждой вершины $v \in V$ находить величину $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

24.1.6 ★

Предположим, что взвешенный ориентированный граф $G = (V, E)$ содержит цикл с отрицательным весом. Разработайте эффективный алгоритм, позволяющий вывести список вершин одного такого цикла. Докажите, что предложенный вами алгоритм корректен.

24.2. Кратчайшие пути из одной вершины в ориентированных ациклических графах

Ослабляя ребра взвешенного ориентированного ациклического графа $G = (V, E)$ в порядке, определенном топологической сортировкой его вершин, кратчайшие пути из одной вершины можно найти за время $\Theta(V + E)$. В ориентированном ациклическом графе кратчайшие пути всегда вполне определены, поскольку, даже если вес некоторых ребер отрицателен, циклов с отрицательными весами не существует.

Работа алгоритма начинается с топологической сортировки ориентированного ациклического графа (см. раздел 22.4), которая должна установить линейное упорядочение вершин. Если путь из вершины u к вершине v существует, то в топологической сортировке вершина u предшествует вершине v. По вершинам, расположенным в топологическом порядке, проход выполняется только один раз. При обработке каждой вершины производится ослабление всех ребер, исходящих из этой вершины.

```
DAG-SHORTEST-PATHS(G, w, s)
1  Топологическая сортировка вершин графа G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for каждой вершины u в порядке топологической сортировки
4      for каждой вершины v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Работа этого алгоритма проиллюстрирована на рис. 24.5.

Время работы этого алгоритма легко проанализировать. Как показано в разделе 22.4, топологическая сортировка в строке 1 выполняется за время $\Theta(V + E)$. Вызов процедуры INITIALIZE-SINGLE-SOURCE в строке 2 занимает время $\Theta(V)$. На каждую вершину приходится по одной итерации цикла **for** в строках 3–5. Цикл **for** в строках 4 и 5 ослабляет каждое ребро по одному разу (мы используем здесь групповой анализ). Поскольку каждая итерация внутреннего цикла **for** занимает время $\Theta(1)$, полное время работы алгоритма равно $\Theta(V + E)$, т.е. линейно зависит от размера представления графа с использованием списков смежности.

Из сформулированной далее теоремы видно, что процедура DAG-SHORTEST-PATHS корректно вычисляет кратчайшие пути.

Теорема 24.5

Если взвешенный ориентированный граф $G = (V, E)$ имеет исток s и не содержит циклов, то по завершении процедуры DAG-SHORTEST-PATHS для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$, а подграф предшествования G_π представляет собой дерево кратчайших путей.

Доказательство. Сначала покажем, что по завершении процедуры для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$. Если вершина v недостижима из истока s, то согласно свойству отсутствия пути выполняется соотноше-

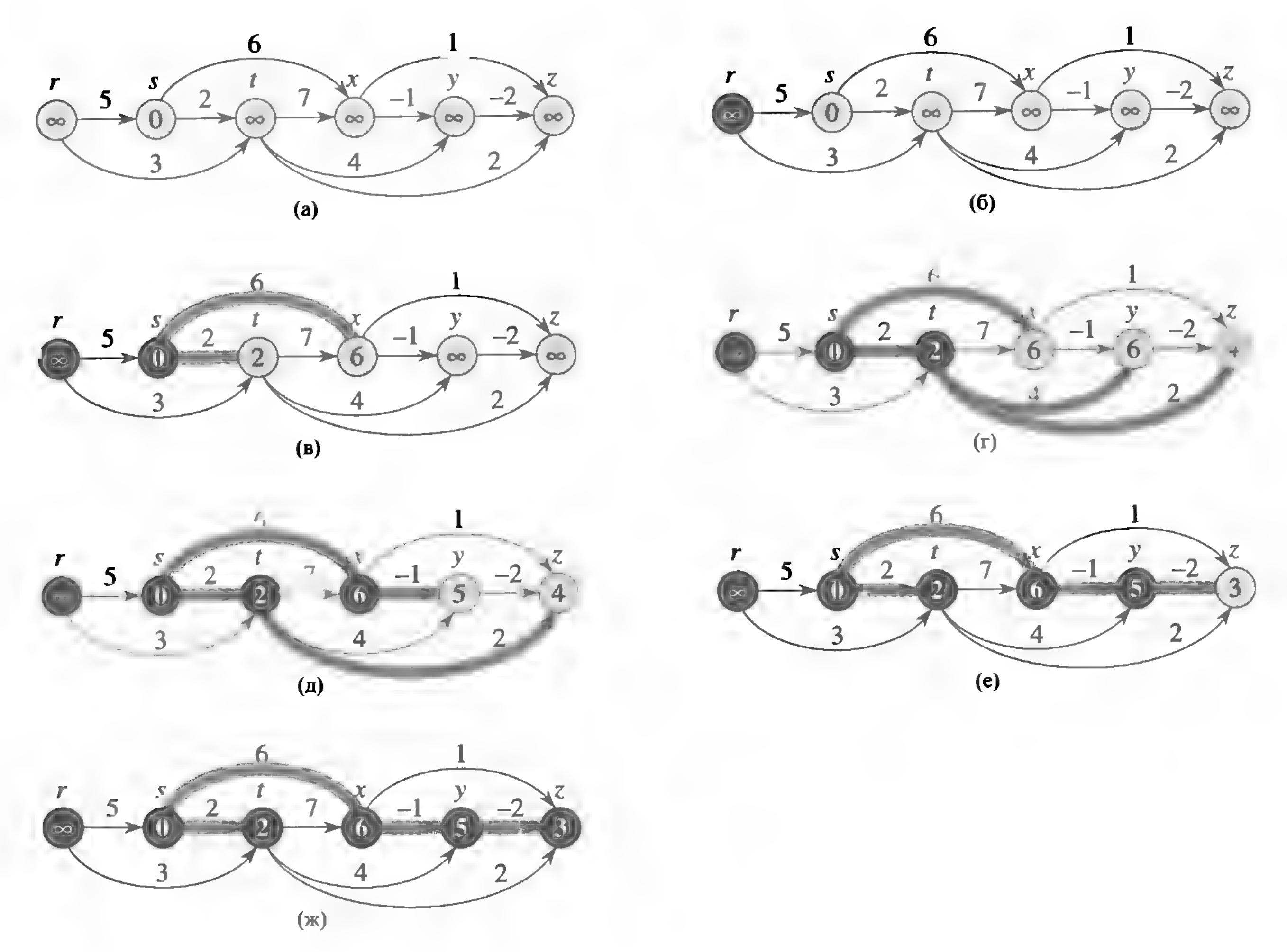


Рис. 24.5. Работа алгоритма поиска кратчайших путей в ориентированном ациклическом графе. Вершины топологически отсортированы слева направо. Истоком является вершина s. В вершинах указаны значения атрибута d, а ребра указывают значения π. **(а)** Ситуация перед первой итерацией цикла **for** в строках 3–5. **(б)–(ж)** Ситуация после каждой итерации цикла **for** в строках 3–5. Вновь зачерненная на очередной итерации вершина используется в качестве вершины u. Значения, показанные в части (ж), являются окончательными.

ние $v.d = \delta(s, v) = \infty$. Теперь предположим, что вершина v достижима из истока s, а следовательно, существует кратчайший путь $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$, а $v_k = v$. Поскольку вершины обрабатываются в порядке топологической сортировки, ребра пути p ослабляются в порядке $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Из свойства ослабления путей следует, что по завершении процедуры для всех $i = 0, 1, \ldots, k$ выполняется равенство $v_i.d = \delta(s, v_i)$. И наконец согласно свойству подграфа предшествования G_π является деревом кратчайших путей. ∎

Интересное применение этого алгоритма возникает при определении критических путей во время анализа ***диаграммы PERT*** (PERT chart)[2]. Ее ребра представляют предназначенные для выполнения задания, а вес каждого из ребер — промежутки времени, необходимые для выполнения того или иного задания. Ес-

[2]Аббревиатура "PERT" расшифровывается как "program evaluation and review technique" — система планирования и руководства разработками.

ли ребро (u, v) входит в вершину v, а ребро (v, x) покидает ее, это означает, что задание (u, v) должно быть выполнено до задания (v, x). Путь по такому ориентированному ациклическому графу представляет последовательность заданий, которые необходимо выполнить в определенном порядке. ***Критический путь*** (critical path) — *самый длинный* путь в ориентированном ациклическом графе, соответствующий самому длительному времени, необходимому для выполнения упорядоченной последовательности заданий. Таким образом, вес критического пути предоставляет нижнюю границу полного времени выполнения всех заданий. Критический путь можно найти одним из следующих двух способов:

- заменив знаки всех весов ребер и выполнив алгоритм DAG-SHORTEST-PATHS;
- воспользовавшись модифицированным алгоритмом DAG-SHORTEST-PATHS, в котором в строке 2 процедуры INITIALIZE-SINGLE-SOURCE значение ∞ заменено значением $-\infty$, а в процедуре RELAX знак "$>$" заменен знаком "$<$".

Упражнения

24.2.1

Выполните процедуру DAG-SHORTEST-PATHS для графа, показанного на рис. 24.5, с вершиной r в качестве истока.

24.2.2

Предположим, что строка 3 в процедуре DAG-SHORTEST-PATHS изменена следующим образом:

3 **for** первых $|V| - 1$ вершин в порядке топологической сортировки

Покажите, что процедура останется корректной.

24.2.3

Представленное выше описание диаграммы PERT несколько неестественно. Естественнее было бы, если бы вершины представляли задания, а ребра — ограничения, накладываемые на порядок их выполнения, т.е. если бы наличие ребра (u, v) указывало на то, что задание u необходимо выполнить до задания v. Тогда веса присваивались бы не ребрам, а вершинам. Модифицируйте процедуру DAG-SHORTEST-PATHS таким образом, чтобы она за линейное время позволяла найти самый длинный путь в ориентированном ациклическом графе со взвешенными вершинами.

24.2.4

Разработайте эффективный алгоритм, позволяющий определить общее количество путей, содержащихся в ориентированном ациклическом графе. Проанализируйте время работы этого алгоритма.

24.3. Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу поиска кратчайших путей из одной вершины во взвешенном ориентированном графе $G = (V, E)$ в случае, когда веса ребер неотрицательны. Поэтому в настоящем разделе предполагается, что для всех ребер $(u, v) \in E$ выполняется неравенство $w(u, v) \geq 0$. Через некоторое время станет понятно, что при хорошей реализации время работы алгоритма Дейкстры меньше времени работы алгоритма Беллмана–Форда.

В алгоритме Дейкстры поддерживается множество вершин S, для которых уже вычислены окончательные веса кратчайших путей к ним из истока s. В этом алгоритме поочередно выбирается вершина $u \in V - S$, которой на данном этапе соответствует минимальная оценка кратчайшего пути. После добавления этой вершины u в множество S проводится ослабление всех исходящих из нее ребер. В приведенной ниже реализации используется неубывающая очередь с приоритетами Q, состоящая из вершин, в роли ключей которых выступают значения их атрибутов d.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for каждой вершины v ∈ G.Adj[u]
8          RELAX(u, v, w)
           // С соответствующими вызовами DECREASE-KEY
```

Процесс ослабления ребер в алгоритме Дейкстры проиллюстрирован на рис. 24.6. В строке 1 выполняется обычная инициализация величин d и π, а в строке 2 инициализируется пустое множество вершин S. В этом алгоритме поддерживается инвариант, согласно которому в начале каждой итерации цикла **while** в строках 4–8 выполняется равенство $Q = V - S$. В строке 3 неубывающая очередь с приоритетами Q инициализируется таким образом, чтобы она содержала все вершины множества V; поскольку в этот момент $S = \emptyset$, после выполнения строки 3 сформулированный выше инвариант выполняется. На каждой итерации цикла **while** в строках 4–8 в строке 5 вершина u извлекается из множества $Q = V - S$ и добавляется в множество S в строке 6, в результате чего инвариант продолжает соблюдаться. (Во время первой итерации этого цикла $u = s$.) Таким образом, вершина u имеет наименьшую оценку кратчайшего пути среди всех вершин множества $V - S$. Затем в строках 7 и 8 ослабляются все ребра (u, v), исходящие из вершины u. Если текущий кратчайший путь к вершине v может быть улучшен в результате прохождения через вершину u, выполняется ослабление и соответствующее обновление оценки $v.d$ и предшественника $v.\pi$. Обратите внимание.

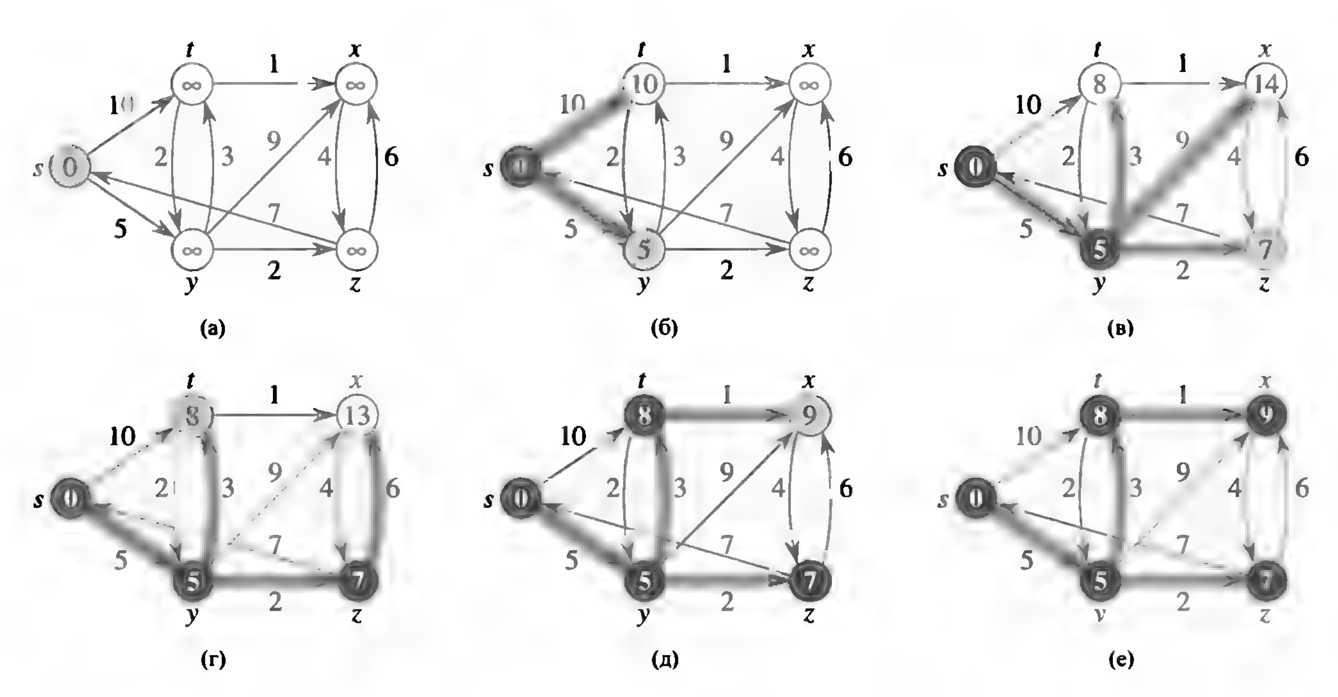


Рис. 24.6. Работа алгоритма Дейкстры. Истоком s является крайняя слева вершина. В вершинах показаны оценки кратчайших путей, а заштрихованные ребра указывают предшественников. Черные вершины находятся в множестве S, а белые — в неубывающей очереди с приоритетами $Q = V - S$. **(а)** Ситуация непосредственно перед первой итерацией цикла **while** в строках 4–8. Заштрихованная вершина имеет минимальное значение d и выбирается в качестве вершины u в строке 5. **(б)–(е)** Ситуация после каждой последующей итерации цикла **while**. В каждой части в очередной итерации в качестве вершины u в строке 5 выбирается заштрихованная вершина. Значения d и предшественники, показанные в части (е), являются окончательными.

что после выполнения строки 3 вершины никогда не добавляются в множество Q и что каждая вершина извлекается из этого множества и добавляется в множество S ровно по одному разу, поэтому количество итераций цикла **while** в строках 4–8 составляет в точности $|V|$.

Поскольку в алгоритме Дейкстры из множества $V - S$ для помещения в множество S всегда выбирается самая "легкая", или "близкая", вершина, можно утверждать, что этот алгоритм придерживается жадной стратегии. Жадные стратегии подробно описаны в главе 16, однако для понимания принципа работы алгоритма Дейкстры читать эту главу необязательно. Жадные стратегии не всегда приводят к оптимальным результатам, однако, как видно из приведенной ниже теоремы и следствия из нее, алгоритм Дейкстры действительно находит кратчайшие пути. Главное — показать, что после каждого добавления вершины u в множество S выполняется равенство $u.d = \delta(s, u)$.

Теорема 24.6 (Корректность алгоритма Дейкстры)

По завершении обработки алгоритмом Дейкстры взвешенного ориентированного графа $G = (V, E)$ с неотрицательной весовой функцией w и истоком s для всех вершин $u \in V$ выполняется равенство $u.d = \delta(s, u)$.

Доказательство. Воспользуемся следующим инвариантом цикла:

> В начале каждой итерации цикла **while** в строках 4–8 для каждой вершины $v \in S$ выполняется равенство $v.d = \delta(s, v)$.

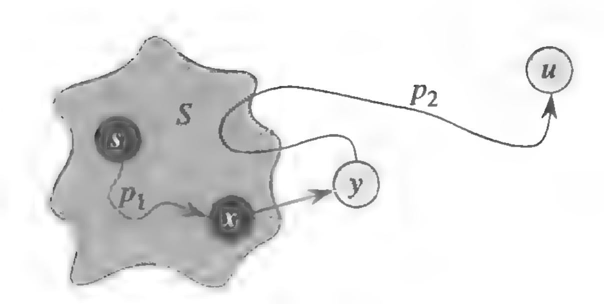


Рис. 24.7. Доказательство теоремы 24.6. Множество S перед добавлением в него вершины u непустое. Разделим кратчайший путь p из истока s к вершине u на $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, где y — первая вершина на пути, не входящая в S, а $x \in S$ — непосредственный предшественник y. Вершины x и y различны, но может быть так, что $s = x$ или $y = u$. Путь p_2 может (необязательно) повторно входить в S.

Достаточно показать, что для каждой вершины $u \in V$ равенство $u.d = \delta(s, u)$ выполняется в момент ее добавления в множество S. После того как будет показана справедливость равенства $u.d = \delta(s, u)$, на основе свойства верхней границы мы покажем, что это равенство продолжает выполняться и в дальнейшем.

Инициализация. Изначально $S = \emptyset$, так что инвариант тривиально выполняется.

Сохранение. Нужно показать, что при каждой итерации равенство $u.d = \delta(s, u)$ выполняется для каждой вершины, добавленной в множество S. Воспользуемся методом "от противного". Чтобы получить противоречие, предположим, что u — первая добавленная в множество S вершина, для которой $u.d \neq \delta(s, u)$. Сосредоточим внимание на ситуации, сложившейся в начале той итерации цикла **while**, в которой вершина u добавляется в множество S. Проанализировав кратчайший путь из вершины s в вершину u, можно будет получить противоречие, заключающееся в том, что в тот момент справедливо равенство $u.d = \delta(s, u)$. Должно выполняться условие $u \neq s$, поскольку s — первая вершина, добавленная в множество S, и в момент ее добавления выполняется равенство $s.d = \delta(s, s) = 0$. Из условия $u \neq s$ следует также, что непосредственно перед добавлением вершины u в множество S оно не является пустым. Из вершины s в вершину u должен вести какой-нибудь путь, так как в противном случае, в соответствии со свойством отсутствия пути, выполняется соотношение $u.d = \delta(s, u) = \infty$, нарушающее справедливость предположения, что $u.d \neq \delta(s, u)$. Поскольку хоть один путь существует, должен существовать и кратчайший путь p из вершины s в вершину u. Перед добавлением вершины u в множество S путь p соединяет вершину из множества S (а именно — вершину s) с вершиной из множества $V - S$ (а именно — с вершиной u). Рассмотрим первую вершину y на пути p, принадлежащую множеству $V - S$, а также предположим, что на этом пути ей предшествует вершина $x \in S$. Тогда, как видно из рис. 24.7, путь p можно разложить на составляющие $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$. (Любой из путей p_1 и p_2 может не содержать ни одного ребра.)

Мы утверждаем, что $y.d = \delta(s, y)$ при добавлении u в S. Для доказательства этого утверждения заметим, что $x \in S$. Затем, поскольку мы выбираем u как первую вершину, для которой $u.d \neq \delta(s, u)$ при ее добавлении в S, при

добавлении x в S мы имеем $x.d = \delta(s, x)$. Ребро (x, y) в этот момент было ослаблено, и наше утверждение вытекает из свойства сходимости.

Теперь можно получить противоречие, позволяющее доказать, что $u.d = \delta(s, u)$. Поскольку на кратчайшем пути из вершины s в вершину u вершина y находится перед вершиной u и вес каждого из ребер выражается неотрицательным значением (это особенно важно для ребер, из которых состоит путь p_2), выполняется неравенство $\delta(s, y) \le \delta(s, u)$, так что

$$\begin{aligned} y.d &= \delta(s, y) \\ &\le \delta(s, u) \\ &\le u.d \qquad \text{(согласно свойству верхней границы)}\ . \end{aligned} \tag{24.2}$$

Однако поскольку и вершина u, и вершина y во время выбора вершины u в строке 5 находились в множестве $V - S$, выполняется неравенство $u.d \le y.d$. Таким образом, оба неравенства в соотношении (24.2) фактически являются равенствами, т.е.

$$y.d = \delta(s, y) = \delta(s, u) = u.d\ .$$

Следовательно, $u.d = \delta(s, u)$, что противоречит нашему выбору вершины u. Мы приходим к выводу, что во время добавления вершины u в множество S выполняется равенство $u.d = \delta(s, u)$, а следовательно, оно выполняется и в дальнейшем.

Завершение. По завершении работы алгоритма $Q = \emptyset$. Из этого факта и инварианта $Q = V - S$ следует, что $S = V$. Таким образом, для всех вершин $u \in V$ выполняется равенство $u.d = \delta(s, u)$. ■

Следствие 24.7

Если выполнить алгоритм Дейкстры для взвешенного ориентированного графа $G = (V, E)$ с неотрицательной весовой функцией w и истоком s, то по завершении работы алгоритма подграф предшествования G_π является деревом кратчайших путей с корнем в вершине s.

Доказательство. Сформулированное выше утверждение непосредственно следует из теоремы 24.6 и из свойства подграфа предшествования. ■

Анализ алгоритма

Насколько быстро работает алгоритм Дейкстры? В нем поддерживается неубывающая очередь с приоритетами Q и тремя операциями, характерными для очередей с приоритетами: INSERT (явно вызывается в строке 3), EXTRACT-MIN (строка 5) и DECREASE-KEY (неявно присутствует в процедуре RELAX, которая вызывается в строке 8). Процедура INSERT, как и процедура EXTRACT-MIN, вызывается по одному разу для каждой вершины. Поскольку каждая вершина $u \in V$ добавляется в множество S ровно по одному разу, каждое ребро в списке смежности $Adj[u]$ обрабатывается в цикле **for** в строках 7 и 8 ровно по одному разу на протяжении работы алгоритма. Так как общее количество ребер во всех списках

смежности равно $|E|$, всего выполняется $|E|$ итераций этого цикла **for**, а следовательно, не более $|E|$ вызовов DECREASE-KEY. (Еще раз заметим, что здесь используется групповой анализ.)

Время работы алгоритма Дейкстры зависит от реализации неубывающей очереди с приоритетами. Сначала рассмотрим случай, когда неубывающая очередь с приоритетами поддерживается за счет того, что все вершины пронумерованы от 1 до $|V|$. Атрибут $v.d$ просто помещается в элемент массива с индексом v. Каждая операция INSERT и DECREASE-KEY занимает время $O(1)$, а каждая операция EXTRACT-MIN — время $O(V)$ (поскольку в ней выполняется поиск по всему массиву); в результате полное время работы алгоритма равно $O(V^2 + E) = O(V^2)$.

Если граф достаточно разреженный, в частности, если количество вершин и ребер в нем связаны соотношением $E = o(V^2/\lg V)$, алгоритм можно сделать более эффективным путем реализации неубывающей очереди с приоритетами с помощью бинарной неубывающей пирамиды. (Как говорится в разделе 6.5, важная деталь реализации заключается в том, что вершины и соответствующие им элементы пирамиды должны поддерживать идентификаторы друг друга.) Каждая операция EXTRACT-MIN занимает время $O(\lg V)$. Как и ранее, всего таких операций — $|V|$. Время, необходимое для построения неубывающей пирамиды, равно $O(V)$. Каждая операция DECREASE-KEY выполняется за время $O(\lg V)$, а всего таких операций выполняется не более $|E|$. Поэтому полное время работы алгоритма составляет $O((V + E)\lg V)$, что равно $O(E \lg V)$, если все вершины достижимы из истока. Это время работы оказывается лучшим по сравнению со временем работы прямой реализации $O(V^2)$, если $E = o(V^2/\lg V)$.

Фактически можно достичь времени работы алгоритма $O(V \lg V + E)$, если неубывающая очередь с приоритетами реализуется с помощью фибоначчиевой пирамиды (см. главу 19). Амортизированная стоимость каждой из $|V|$ операций EXTRACT-MIN равна $O(\lg V)$, а каждый вызов процедуры DECREASE-KEY (всего их не более $|E|$) требует лишь $O(1)$ амортизированного времени. Исторически сложилось так, что развитие пирамид Фибоначчи было стимулировано наблюдением, согласно которому в алгоритме Дейкстры процедура DECREASE-KEY обычно вызывается намного чаще, чем процедура EXTRACT-MIN, поэтому любой метод, уменьшающий амортизированное время каждой операции DECREASE-KEY до величины $o(\lg V)$, не увеличивая при этом амортизированного времени операции EXTRACT-MIN, позволяет получить реализацию, которая в асимптотическом пределе работает быстрее, чем реализация с помощью бинарных пирамид.

Алгоритм Дейкстры имеет некоторую схожесть как с поиском в ширину (см. раздел 22.2), так и с алгоритмом Прима для построения минимальных остовных деревьев (см. раздел 23.2). Поиск в ширину он напоминает в том отношении, что множество S соответствует множеству черных вершин, используемых при поиске в ширину; точно так же, как вершинам множества S сопоставляются окончательные веса кратчайших путей, так и черным вершинам при поиске в ширину сопоставляются правильные расстояния. На алгоритм Прима алгоритм Дейкстры похож в том плане, что в обоих алгоритмах с помощью неубывающей очереди с приоритетами находится "легчайшая" вершина за пределами заданного множества (в алгоритме Дейкстры это множество S, а в алгоритме Прима —

выращиваемое остовное дерево), затем эта вершина добавляется в множество, после чего соответствующим образом корректируются и упорядочиваются веса оставшихся за пределами множества вершин.

Упражнения

24.3.1

Выполните алгоритм Дейкстры над ориентированным графом, изображенным на рис. 24.2. Рассмотрите два варианта задачи, в одном из которых истоком является вершина s, а в другом — вершина z. Используя в качестве образца рис. 24.6, укажите, чему будут равны значения d и π и какие вершины будут входить в множество S после каждой итерации цикла **while**.

24.3.2

Приведите простой пример ориентированного графа с отрицательными весами ребер, для которого алгоритм Дейкстры дает неправильные ответы. Почему доказательство теоремы 24.6 не годится, если веса ребер могут быть отрицательными?

24.3.3

Предположим, что мы изменяем строку 4 алгоритма Дейкстры следующим образом.

4 **while** $|Q| > 1$

В результате этого изменения цикл **while** выполняется $|V| - 1$ раз, а не $|V|$ раз. Корректен ли предложенный алгоритм?

24.3.4

Профессор написал программу, которая, как он утверждает, реализует алгоритм Дейкстры. Программа генерирует значения $v.d$ и $v.\pi$ для каждой вершины $v \in V$. Приведите алгоритм со временем работы $O(V + E)$, проверяющий выход программы профессора. Он должен определять, соответствуют ли атрибуты d и π атрибутам некоторого дерева кратчайших путей. Вы можете считать, что веса всех ребер неотрицательны.

24.3.5

Профессор думает, что он разработал более простое доказательство корректности алгоритма Дейкстры. Он утверждает, что алгоритм Дейкстры ослабляет ребра на каждом кратчайшем пути в графе в том порядке, в котором они находятся в пути, а следовательно, свойство ослабления пути применимо к каждой вершине, достижимой из истока. Покажите, что профессор ошибается, построив ориентированный граф, для которого алгоритм Дейкстры ослабляет ребра кратчайшего пути не по порядку.

24.3.6

Пусть дан ориентированный граф $G = (V, E)$, с каждым ребром $(u, v) \in E$ которого связано значение $r(u, v)$, являющееся действительным числом в интерва-

ле $0 \leq r(u, v) \leq 1$ и представляющее надежность соединительного кабеля между вершиной u и вершиной v. Величина $r(u, v)$ интерпретируется как вероятность того, что в канале, соединяющем вершины u и v, не произойдет сбой; при этом предполагается, что все вероятности независимы. Сформулируйте эффективный алгоритм, позволяющий найти наиболее надежный путь, соединяющий две заданные вершины.

24.3.7

Пусть $G = (V, E)$ — взвешенный ориентированный граф с положительной весовой функцией $w : E \to \{1, 2, \ldots, W\}$, где W — некоторое положительное целое число. Будем считать, что ни для каких двух вершин кратчайшие пути, ведущие из истока s в эти вершины, не имеют одинаковых весов. Предположим, что определен невзвешенный ориентированный граф $G' = (V \cup V', E')$, в котором каждое ребро $(u, v) \in E$ заменяется $w(u, v)$ последовательными ребрами единичного веса. Сколько вершин содержит граф G'? Теперь предположим, что в графе G' выполняется поиск в ширину. Покажите, что порядок, в котором вершины множества V окрашиваются в черный цвет при поиске в ширину в графе G', совпадает с порядком извлечения вершин множества V из очереди с приоритетами при выполнении алгоритма Дейкстры над графом G.

24.3.8

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \{0, 1, \ldots, W\}$, где W — некоторое целое неотрицательное число. Модифицируйте алгоритм Дейкстры так, чтобы он вычислял кратчайшие пути из заданной вершины s за время $O(WV + E)$.

24.3.9

Модифицируйте алгоритм из упр. 24.3.8 таким образом, чтобы он выполнялся за время $O((V + E) \lg W)$. (*Указание:* сколько различных оценок кратчайших путей для вершин из множества $V - S$ может встретиться одновременно?)

24.3.10

Предположим, что имеется взвешенный ориентированный граф $G = (V, E)$, в котором веса ребер, исходящих из некоторого истока s, могут быть отрицательными, веса всех других ребер неотрицательные, а циклы с отрицательными весами отсутствуют. Докажите, что в таком графе алгоритм Дейкстры корректно находит кратчайшие пути из истока s.

24.4. Разностные ограничения и кратчайшие пути

В главе 29 изучается общая задача линейного программирования, в которой нужно оптимизировать линейную функцию, удовлетворяющую системе линейных неравенств. В этом разделе исследуется частный случай задачи линейного программирования, который сводится к поиску кратчайших путей из одной вер-

шины. Полученную в результате задачу о кратчайших путях из одной вершины можно решить с помощью алгоритма Беллмана–Форда, решив таким образом задачу линейного программирования.

Линейное программирование

В обобщенной ***задаче линейного программирования*** (linear-programming problem) задаются матрица A размером $m \times n$, m-компонентный вектор b и n-компонентный вектор c. Нужно найти состоящий из n элементов вектор x, максимизирующий ***целевую функцию*** (objective function) $\sum_{i=1}^{n} c_i x_i$, на которую накладывается m ограничений $Ax \le b$.

Несмотря на то что время работы симплекс-алгоритма, который рассматривается в главе 29, не всегда является полиномиальной функцией от размера входных данных, существуют другие алгоритмы линейного программирования с полиномиальным временем работы. Имеется несколько причин, по которым важно понимать, как устроены задачи линейного программирования. Во-первых, если известно, что некоторая задача приводится к задаче линейного программирования с полиномиальным размером, то отсюда непосредственно следует, что для такой задачи существует алгоритм с полиномиальным временем работы. Во-вторых, имеется большое количество частных случаев задач линейного программирования, для которых существуют более производительные алгоритмы. Например, задача о кратчайшем пути между парой заданных вершин (упр. 24.4.4) и задача о максимальном потоке (упр. 26.1.5) являются частными случаями задачи линейного программирования.

Иногда не имеет значения, какой вид имеет целевая функция; достаточно найти произвольное ***допустимое решение*** (feasible solution), т.е. любой вектор x, удовлетворяющий неравенству $Ax \le b$, или убедиться, что допустимых решений не существует. Сосредоточим внимание на таких ***задачах существования*** (feasibility problem).

Системы разностных ограничений

В ***системе разностных ограничений*** (system of difference constraints) каждая строка в матрице линейного программирования A содержит одно значение 1 и одно значение -1, а все прочие элементы в этой строке равны 0. Другими словами, ограничения, заданные системой неравенств $Ax \le b$, представляют собой систему m ***разностных ограничений*** (difference constraints), содержащих n неизвестных. Каждое ограничение в этой системе — обычное линейное неравенство вида

$$x_j - x_i \le b_k ,$$

где $1 \le i, j \le n$, $i \ne j$ и $1 \le k \le m$.

Рассмотрим, например, задачу поиска 5-компонентного вектора $x = (x_i)$, удовлетворяющего системе неравенств

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

Эта задача эквивалентна поиску неизвестных величин x_1, x_2, x_3, x_4, x_5, удовлетворяющих восьми разностным ограничениям:

$$x_1 - x_2 \leq 0 , \tag{24.3}$$
$$x_1 - x_5 \leq -1 , \tag{24.4}$$
$$x_2 - x_5 \leq 1 , \tag{24.5}$$
$$x_3 - x_1 \leq 5 , \tag{24.6}$$
$$x_4 - x_1 \leq 4 , \tag{24.7}$$
$$x_4 - x_3 \leq -1 , \tag{24.8}$$
$$x_5 - x_3 \leq -3 , \tag{24.9}$$
$$x_5 - x_4 \leq -3 . \tag{24.10}$$

Одним из решений этой задачи является $x = (-5, -3, 0, -1, -4)$, что можно проверить прямой подстановкой. На самом деле эта задача имеет множество решений. Например, еще одно решение — $x' = (0, 2, 5, 4, 1)$. Эти два решения взаимосвязаны: разность между любой парой соответствующих компонентов векторов x' и x равна 5. Этот факт — не простое совпадение.

Лемма 24.8

Пусть $x = (x_1, x_2, \ldots, x_n)$ является решением системы разностных ограничений $Ax \leq b$, а d — произвольная константа. Тогда $x+d = (x_1+d, x_2+d, \ldots, x_n+d)$ также является решением системы $Ax \leq b$.

Доказательство. Для каждой пары переменных x_i и x_j можно записать соотношение $(x_i + d) - (x_j + d) = x_i - x_j$. Таким образом, если вектор x удовлетворяет системе неравенств $Ax \leq b$, то ей удовлетворяет и вектор $x + d$. ∎

Системы разностных ограничений возникают в самых разнообразных приложениях. Например, неизвестные величины x_i могут обозначать моменты времени, в которые происходят события. Каждое ограничение можно рассматривать как требование, при котором между двумя событиями должно пройти некоторое время, не меньшее (или не большее) некоторой заданной величины. Возможно, эти события — задания, которые необходимо выполнить в процессе сборки неко-

торого изделия. Например, если в момент времени x_1 применяется клей, время фиксации которого — 2 часа, а деталь будет устанавливаться в момент времени x_2, то необходимо наложить ограничение $x_2 \geq x_1 + 2$, или, что эквивалентно, $x_1 - x_2 \leq -2$. При другой технологии может потребоваться, чтобы деталь устанавливалась после применения клея, но не позже, чем когда он "схватится" наполовину. В этом случае получаем пару ограничений $x_2 \geq x_1$ и $x_2 \leq x_1 + 1$, или, что эквивалентно, $x_1 - x_2 \leq 0$ и $x_2 - x_1 \leq 1$.

Графы ограничений

Системы разностных ограничений можно рассматривать с точки зрения теории графов. Идея заключается в том, что в системе разностных ограничений $Ax \leq b$ матрицу задачи линейного программирования A размером $m \times n$ можно рассматривать как транспонированную матрицу инцидентности (см. упр. 22.1.7) графа с n вершинами и m ребрами. Каждая вершина v_i графа при $i = 1, 2, \ldots, n$ соответствует одной из n неизвестных переменных x_i. Каждое ориентированное ребро графа соответствует одному из m независимых неравенств, в которое входят две неизвестные величины.

Если выражаться формальным языком, то заданной системе разностных ограничений $Ax \leq b$ сопоставляется ***граф ограничений*** (constraint graph), представляющий собой взвешенный ориентированный граф $G = (V, E)$, в котором

$$V = \{v_0, v_1, \ldots, v_n\}$$

и

$$\begin{aligned} E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ является ограничением}\} \\ \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} \ . \end{aligned}$$

Вскоре станет понятно, что дополнительная вершина v_0 вводится для того, чтобы в графе имелась вершина, из которой гарантированно была достижима любая другая вершина. Таким образом, множество V состоит из вершин v_i, каждая из которых соответствует неизвестной x_i, и дополнительной вершины v_0. В множестве ребер E на каждое разностное ограничение приходится по одному ребру; кроме того, в это множество входят ребра (v_0, v_i), каждое из которых соответствует неизвестной величине x_i. Если накладывается разностное ограничение $x_j - x_i \leq b_k$, это означает, что вес ребра (v_i, v_j) равен $w(v_i, v_j) = b_k$. Вес каждого ребра, исходящего из вершины v_0, равен 0. На рис. 24.8 приведен граф ограничений для системы разностных ограничений (24.3)–(24.10).

Из приведенной ниже теоремы видно, что решение системы разностных ограничений можно найти путем определения весов кратчайших путей в соответствующем графе ограничений.

Теорема 24.9

Пусть $G = (V, E)$ представляет собой граф ограничений, соответствующий заданной системе разностных ограничений $Ax \leq b$. Если граф G не содержит циклов

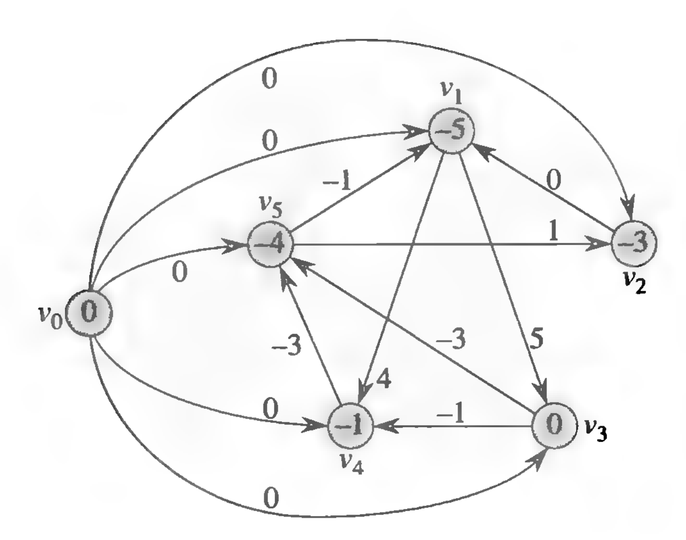


Рис. 24.8. Граф ограничений для системы разностных ограничений (24.3)–(24.10). В каждой вершине v_i указано значение $\delta(v_0, v_i)$. Одно допустимое решение этой системы имеет вид $x = (-5, -3, 0, -1, -4)$.

с отрицательным весом, то вектор

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)) \tag{24.11}$$

является допустимым решением системы. Если граф G содержит циклы с отрицательным весом, то допустимых решений не существует.

Доказательство. Сначала покажем, что если граф ограничений не содержит циклов с отрицательным весом, то уравнение (24.11) дает допустимое решение. Рассмотрим произвольное ребро $(v_i, v_j) \in E$. Согласно неравенству треугольника выполняется неравенство $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ или, что то же самое, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Таким образом, полагая $x_i = \delta(v_0, v_i)$ и $x_j = \delta(v_0, v_j)$, мы удовлетворяем разностное ограничение $x_j - x_i \leq w(v_i, v_j)$, соответствующее ребру (v_i, v_j).

Теперь покажем, что если граф ограничений содержит цикл с отрицательным весом, то система разностных ограничений не имеет допустимых решений. Не теряя общности, предположим, что цикл с отрицательным весом — это цикл $c = \langle v_1, v_2, \ldots, v_k \rangle$, где $v_1 = v_k$. (Вершина v_0 не может принадлежать циклу c, потому что в нее не входит ни одно ребро.) Цикл c соответствует приведенной ниже системе ограничений:

$$\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2) \;, \\
x_3 - x_2 &\leq w(v_2, v_3) \;, \\
&\;\;\vdots \\
x_{k-1} - x_{k-2} &\leq w(v_{k-2}, v_{k-1}) \;, \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k) \;.
\end{aligned}$$

Предположим, что существует решение x, удовлетворяющее всем этим k неравенствам. Это решение должно также удовлетворять неравенству, полученному

в результате суммирования всех перечисленных k неравенств. Если просуммировать все левые части, то каждая неизвестная x_i один раз войдет со знаком "+" и один раз — со знаком "−" (напомним, что из равенства $v_1 = v_k$ следует равенство $x_1 = x_k$). Это означает, что сумма всех левых частей равна 0. Сумма правых частей равна $w(c)$, откуда получается неравенство $0 \leq w(c)$. Однако, поскольку c — цикл с отрицательным весом, выполняется неравенство $w(c) < 0$, что приводит нас к противоречию $0 \leq w(c) < 0$. ■

Решение систем разностных ограничений

Теорема 24.9 говорит о том, что с помощью алгоритма Беллмана–Форда можно решить систему линейных ограничений. Поскольку в графе ограничений существуют ребра, ведущие из вершины v_0 во все другие вершины, любой цикл с отрицательным весом в графе ограничений достижим из вершины v_0. Если алгоритм Беллмана–Форда возвращает значение TRUE, веса кратчайших путей образуют допустимое решение этой системы. В примере, проиллюстрированном на рис. 24.8, веса кратчайших путей образуют допустимое решение $x = (-5, -3, 0, -1, -4)$, и согласно лемме 24.8 $x = (d-5, d-3, d, d-1, d-4)$, где d — произвольная константа, также является допустимым решением. Если алгоритм Беллмана–Форда возвращает значение FALSE, то система разностных ограничений решений не имеет.

Система разностных ограничений с m ограничениями и n неизвестными порождает граф, состоящий из $n+1$ вершин и $n+m$ ребер. Таким образом, с помощью алгоритма Беллмана–Форда такую систему можно решить за время $O((n+1)(n+m)) = O(n^2 + nm)$. В упр. 24.4.5 предлагается модифицировать этот алгоритм таким образом, чтобы он выполнялся за время $O(nm)$, даже если m намного меньше, чем n.

Упражнения

24.4.1

Найдите допустимое решение приведенной ниже системы разностных ограничений или покажите, что решений не существует:

$$\begin{aligned}
x_1 - x_2 &\leq 1 , \\
x_1 - x_4 &\leq -4 , \\
x_2 - x_3 &\leq 2 , \\
x_2 - x_5 &\leq 7 , \\
x_2 - x_6 &\leq 5 , \\
x_3 - x_6 &\leq 10 , \\
x_4 - x_2 &\leq 2 , \\
x_5 - x_1 &\leq -1 , \\
x_5 - x_4 &\leq 3 , \\
x_6 - x_3 &\leq -8 .
\end{aligned}$$

24.4.2

Найдите допустимое решение приведенной ниже системы разностных ограничений или покажите, что решений не существует:

$$
\begin{aligned}
x_1 - x_2 &\leq 4 \ , \\
x_1 - x_5 &\leq 5 \ , \\
x_2 - x_4 &\leq -6 \ , \\
x_3 - x_2 &\leq 1 \ , \\
x_4 - x_1 &\leq 3 \ , \\
x_4 - x_3 &\leq 5 \ , \\
x_4 - x_5 &\leq 10 \ , \\
x_5 - x_3 &\leq -4 \ , \\
x_5 - x_4 &\leq -8 \ .
\end{aligned}
$$

24.4.3

Может ли вес любого кратчайшего пути, исходящего из новой вершины v_0 в графе ограничений, быть положительным? Объясните свой ответ.

24.4.4

Сформулируйте задачу о кратчайшем пути между парой заданных вершин в виде задачи линейного программирования.

24.4.5

Покажите, как можно модифицировать алгоритм Беллмана–Форда, чтобы он позволял решить систему разностных ограничений с m неравенствами и n неизвестными за время $O(nm)$.

24.4.6

Предположим, что в дополнение к системе разностных ограничений накладываются ***ограничения-равенства*** (equality constraints), имеющие вид $x_i = x_j + b_k$. Покажите, как адаптировать алгоритм Беллмана–Форда, чтобы с его помощью можно было решать системы ограничений такого вида.

24.4.7

Покажите, как можно решить систему разностных ограничений с помощью алгоритма, подобного алгоритму Беллмана–Форда, который обрабатывает граф ограничений, не содержащий дополнительную вершину v_0.

24.4.8 ★

Пусть $Ax \leq b$ — система m разностных ограничений с n неизвестными. Покажите, что при обработке графа ограничений, соответствующего этой системе, алгоритм Беллмана–Форда максимизирует сумму $\sum_{i=1}^{n} x_i$, где вектор x удовлетворяет системе $Ax \leq b$, а все компоненты x_i — неравенству $x_i \leq 0$.

24.4.9 ★

Покажите, что в процессе обработки графа ограничений, соответствующего системе разностных ограничений $Ax \leq b$, алгоритм Беллмана–Форда минимизирует величину $(\max\{x_i\} - \min\{x_i\})$, где x_i — компоненты вектора x, удовлетворяющего системе $Ax \leq b$. Объясните, как можно использовать этот факт, если алгоритм применяется для составления расписания некоторых конструкторских работ.

24.4.10

Предположим, что каждая строка матрицы A задачи линейного программирования $Ax \leq b$ соответствует либо разностному ограничению, либо ограничению на одну переменную вида $x_i \leq b_k$ или $-x_i \leq b_k$. Покажите, как адаптировать алгоритм Беллмана–Форда для решения систем ограничений такого вида.

24.4.11

Разработайте эффективный алгоритм решения системы разностных ограничений $Ax \leq b$, в которой все элементы вектора b являются действительными числами, а все неизвестные x_i должны быть целыми числами.

24.4.12 ★

Разработайте эффективный алгоритм решения системы разностных ограничений $Ax \leq b$, в которой все элементы вектора b являются действительными числами, а определенное число неизвестных x_i (не обязательно все) должно быть целыми числами.

24.5. Доказательства свойств кратчайших путей

Корректность доказательств, которые используются в этой главе, основана на неравенстве треугольника, свойстве верхней границы, свойстве отсутствия пути, свойстве сходимости, свойстве ослабления пути и свойстве подграфа предшествования. В начале главы эти свойства сформулированы без доказательств. В настоящем разделе мы их докажем.

Неравенство треугольника

При изучении поиска в ширину (раздел 22.2) в лемме 22.1 доказано простое свойство, которым обладают кратчайшие расстояния в невзвешенных графах. Неравенство треугольника обобщает это свойство для взвешенных графов.

Лемма 24.10 (Неравенство треугольника)

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$ и истоком s. Тогда для всех ребер $(u, v) \in E$ выполняется неравенство

$$\delta(s, v) \leq \delta(s, u) + w(u, v) .$$

Доказательство. Предположим, что p представляет собой кратчайший путь из истока s в вершину v. Тогда вес этого пути p не превышает веса любого другого пути из вершины s в вершину v. В частности, вес пути p не превышает веса пути, состоящего из кратчайшего пути из истока s в вершину u и ребра (u, v).

В упр. 24.5.3 предлагается рассмотреть случай, когда не существует кратчайшего пути из вершины s в вершину v. ■

Влияние ослабления на оценки кратчайшего пути

Приведенная ниже группа лемм описывает, какому воздействию подвергаются оценки кратчайшего пути, когда выполняется последовательность шагов ослабления ребер взвешенного ориентированного графа, инициализированного процедурой INITIALIZE-SINGLE-SOURCE.

Лемма 24.11 (Свойство верхней границы)

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Пусть $s \in V$ — исток, а граф инициализирован процедурой INITIALIZE-SINGLE-SOURCE(G, s). Тогда для всех вершин $v \in V$ выполняется неравенство $v.d \geq \delta(s, v)$, и этот инвариант поддерживается в ходе всей последовательности шагов ослабления ребер графа G. Более того, как только атрибут $v.d$ достигает своей нижней границы $\delta(s, v)$, в дальнейшем он не изменяется.

Доказательство. Докажем инвариант $v.d \geq \delta(s, v)$ для всех вершин $v \in V$, воспользовавшись индукцией по числу шагов ослабления.

В качестве базиса индукции используем неравенство $v.d \geq \delta(s, v)$, которое очевидным образом истинно непосредственно после инициализации, поскольку из $v.d = \infty$ вытекает $v.d \geq \delta(s, v)$ для всех $v \in V - \{s\}$ и поскольку $s.d = 0 \geq \delta(s, s)$ (заметим, что $\delta(s, s) = -\infty$, если s находится в цикле с отрицательным весом, и 0 — в противном случае).

В качестве шага индукции рассмотрим ослабление ребра (u, v). Согласно гипотезе индукции для всех вершин $x \in V$ перед ослаблением выполняется неравенство $x.d \geq \delta(s, x)$. Единственное значение d, которое может измениться — это значение $v.d$. Если оно изменяется, мы имеем

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(согласно гипотезе индукции)} \\ &\geq \delta(s, v) && \text{(согласно неравенству треугольника)}, \end{aligned}$$

так что инвариант сохраняется.

Чтобы увидеть, что значение $v.d$ не будет изменяться после того, как станет справедливым соотношение $v.d = \delta(s, v)$, заметим, что по достижении своей нижней границы это значение больше не сможет уменьшаться, поскольку, как уже было показано, $v.d \geq \delta(s, v)$. Оно также не может возрастать, поскольку ослабление не увеличивает значений d. ■

Следствие 24.12 (Свойство отсутствия пути)

Предположим, что во взвешенном ориентированном графе $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ отсутствуют пути, соединяющие исток $s \in V$ с данной вершиной $v \in V$. Тогда после инициализации графа процедурой INITIALIZE-SINGLE-SOURCE(G, s) мы имеем $v.d = \delta(s, v) = \infty$, и это равенство сохраняется в качестве инварианта в ходе выполнения произвольной последовательности шагов ослабления ребер графа G.

Доказательство. Согласно свойству верхней границы мы всегда имеем $\infty = \delta(s, v) \leq v.d$, и, таким образом, $v.d = \infty = \delta(s, v)$. ■

Лемма 24.13

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$ и пусть $(u, v) \in E$. Тогда непосредственно после ослабления ребра (u, v) путем выполнения RELAX(u, v, w) мы имеем $v.d \leq u.d + w(u, v)$.

Доказательство. Если непосредственно перед ослаблением ребра (u, v) мы имеем $v.d > u.d + w(u, v)$, то после ослабления $v.d = u.d + w(u, v)$. Если же перед ослаблением $v.d \leq u.d + w(u, v)$, то ни $u.d$, ни $v.d$ не изменяются, так что после ослабления $v.d \leq u.d + w(u, v)$. ■

Лемма 24.14 (Свойство сходимости)

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$, $s \in V$ — исток, а $s \leadsto u \to v$ — кратчайший путь в графе G для некоторых его вершин $u, v \in V$. Предположим, что граф G инициализирован процедурой INITIALIZE-SINGLE-SOURCE(G, s), после чего выполнена последовательность шагов ослабления, включающая вызов процедуры RELAX(u, v, w). Если в некоторый момент времени до вызова выполняется равенство $u.d = \delta(s, u)$, то в любой момент времени после вызова справедливо равенство $v.d = \delta(s, v)$.

Доказательство. Согласно свойству верхней границы, если в некоторый момент до ослабления ребра (u, v) выполняется равенство $u.d = \delta(s, u)$, то оно остается справедливым и впоследствии. В частности, после ослабления ребра (u, v) мы имеем

$$\begin{aligned} v.d &\leq u.d + w(u, v) && \text{(согласно лемме 24.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(согласно лемме 24.1)} . \end{aligned}$$

Согласно свойству верхней границы $v.d \geq \delta(s, v)$, откуда мы делаем вывод о том, что $v.d = \delta(s, v)$, и это равенство впоследствии сохраняется. ■

Лемма 24.15 (Свойство ослабления пути)

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$, а $s \in V$ — исток. Рассмотрим произвольный кратчай-

ший путь $p = \langle v_0, v_1, \ldots, v_k \rangle$ из истока $s = v_0$ в вершину v_k. Если граф G инициализирован процедурой INITIALIZE-SINGLE-SOURCE(G, s), а затем выполнена последовательность ослаблений ребер $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ в указанном порядке, то после этих ослаблений и в любой момент времени впоследствии выполняется равенство $v_k.d = \delta(s, v_k)$. Это свойство справедливо независимо от того, производятся ли ослабления на других ребрах, включая ослабления, которые чередуются с ослаблениями ребер пути p.

Доказательство. По индукции покажем, что после ослабления i-го ребра пути p выполняется равенство $v_i.d = \delta(s, v_i)$. В качестве базиса примем $i = 0$; перед тем как будет ослаблено хотя бы одно ребро, входящее в путь p, после процедуры инициализации очевидно, что $v_0.d = s.d = 0 = \delta(s, s)$. Согласно свойству верхней границы значение $s.d$ после инициализации больше не изменяется.

На каждом шаге индукции предполагается, что выполняется равенство $v_{i-1}.d = \delta(s, v_{i-1})$, и рассматривается ослабление ребра (v_{i-1}, v_i). Согласно свойству сходимости после этого ослабления выполняется равенство $v_i.d = \delta(s, v_i)$, которое впоследствии сохраняется. ∎

Ослабление и деревья кратчайших путей

Теперь покажем, что после того, как в результате последовательности ослаблений оценки кратчайших путей сойдутся к весам кратчайших путей, подграф предшествования G_π, образованный полученными значениями π, будет деревом кратчайших путей для графа G. Начнем с приведенной ниже леммы, в которой показано, что подграф предшествования всегда образует корневое дерево с корнем в истоке.

Лемма 24.16

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$, а $s \in V$ — исток. Предполагается также, что граф G не содержит циклов с отрицательным весом, достижимых из истока s. Тогда после инициализации графа с помощью процедуры INITIALIZE-SINGLE-SOURCE(G, s) подграф предшествования G_π образует корневое дерево с корнем s, а любая последовательность шагов ослабления ребер графа G поддерживает это свойство в качестве инварианта.

Доказательство. Изначально единственной вершиной графа G_π является исток, и лемма тривиальным образом выполняется. Рассмотрим подграф предшествования G_π, возникающий в результате последовательности шагов ослабления. Сначала докажем, что этот граф ациклический. Чтобы получить противоречие, предположим, что после некоторого шага ослабления в графе G_π создается цикл. Пусть это цикл $c = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_k = v_0$. Тогда для $i = 1, 2, \ldots, k$ выполняется равенство $v_i.\pi = v_{i-1}$, и без потери общности можно считать, что цикл был создан в результате ослабления ребра (v_{k-1}, v_k) графа G_π.

Утверждается, что все вершины цикла c достижимы из истока s. Почему? Для каждой вершины цикла c существует предшественник (т.е. значение соответ-

ствующего атрибута отлично от значения NIL), поэтому каждой из этих вершин сопоставляется конечная оценка кратчайшего пути, когда ее атрибуту π присваивается значение, отличное от значения NIL. Согласно свойству верхней границы, каждой вершине цикла c соответствует вес кратчайшего пути, из чего следует, что она достижима из истока s.

Рассмотрим оценки кратчайших путей, которые относятся к циклу c, непосредственно перед вызовом процедуры RELAX(v_{k-1}, v_k, w) и покажем, что c — цикл с отрицательным весом, а это противоречит предположению о том, что в графе G не содержится циклов с отрицательным весом, достижимых из истока. Непосредственно перед вызовом для $i = 1, 2, \ldots, k-1$ выполняется равенство $v_i.\pi = v_{i-1}$. Таким образом, для $i = 1, 2, \ldots, k-1$ последним обновлением величины $v_i.d$ является присвоение $v_i.d = v_{i-1}.d + w(v_{i-1}, v_i)$. Если после этого значение $v_{i-1}.d$ изменяется, то оно может только уменьшаться. Поэтому непосредственно перед вызовом процедуры RELAX(v_{k-1}, v_k, w) выполняется неравенство

$$v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i) \qquad \text{для всех } i = 1, 2, \ldots, k-1\ . \tag{24.12}$$

Поскольку величина $v_k.\pi$ в результате вызова изменяется, непосредственно перед этим выполняется строгое неравенство

$$v_k.d > v_{k-1}.d + w(v_{k-1}, v_k)\ .$$

Суммируя это строгое неравенство с $k-1$ неравенствами (24.12), получим сумму оценок кратчайшего пути вдоль цикла c:

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &> \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)\ .\end{aligned}$$

Однако

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d\ ,$$

поскольку каждая вершина цикла c входит в каждую сумму ровно по одному разу. Из этого равенства следует

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i)\ .$$

Таким образом, суммарный вес цикла c — величина отрицательная, что и приводит к желаемому противоречию.

Итак, доказано, что G_π — ориентированный ациклический граф. Чтобы показать, что он образует корневое дерево с корнем s, осталось доказать

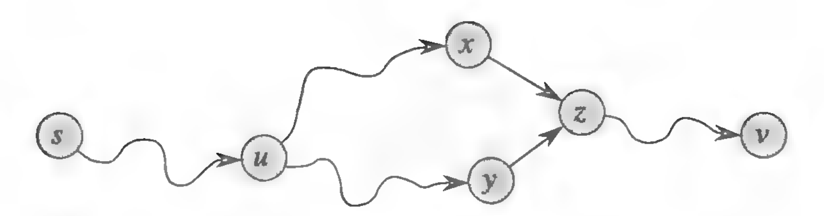


Рис. 24.9. Иллюстрация того, что простой путь в графе G_π из истока s в вершину v — единственный. Если имеются два пути, p_1 ($s \leadsto u \leadsto x \to z \leadsto v$) и p_2 ($s \leadsto u \leadsto y \to z \leadsto v$), где $x \neq y$, то $z.\pi = x$ и $z.\pi = y$, что приводит к противоречию.

(см. упр. Б.5.2), что для каждой вершины $v \in V_\pi$ в графе G_π имеется единственный простой путь из истока s в вершину v.

Сначала необходимо показать, что из истока s существует путь в каждую вершину множества V_π. В это множество входят вершины, значение атрибута π которых отлично от значения NIL, а также вершина s. Идея заключается в том, чтобы доказать наличие такого пути из истока s в каждую вершину множества V_π по индукции. Детали предлагается рассмотреть в упр. 24.5.6.

Чтобы завершить доказательство леммы, теперь нужно показать, что для любой вершины $v \in V_\pi$ в графе G_π существует не более одного пути из вершины s в вершину v. Предположим обратное. Другими словами, предположим, что из истока s существует два простых пути в некоторую вершину v: путь p_1, который можно разложить как $s \leadsto u \leadsto x \to z \leadsto v$, и путь p_2, который можно разложить как $s \leadsto u \leadsto y \to z \leadsto v$, где $x \neq y$ (рис. 24.9). Однако в таком случае выполняются равенства $z.\pi = x$ и $z.\pi = y$, откуда следует противоречие $x = y$. Мы заключаем, что в графе G_π существует единственный путь из истока s в вершину v, а следовательно, этот граф образует корневое дерево с корнем s. ■

Теперь можно показать, что если после некоторой последовательности этапов ослабления всем вершинам присвоены истинные веса кратчайших путей, то подграф предшествования G_π представляет собой дерево кратчайших путей.

Лемма 24.17 (Свойство подграфа предшествования)

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$, а $s \in V$ — исток. Предположим также, что граф G не содержит циклов с отрицательным весом, достижимых из вершины s. Вызовем процедуру INITIALIZE-SINGLE-SOURCE(G, s), после чего выполним произвольную последовательность шагов ослабления ребер графа G, в результате которой для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$. Тогда подграф предшествования G_π является деревом кратчайших путей с корнем s.

Доказательство. Необходимо доказать, что для графа G_π выполняются все три свойства, сформулированные на с. 685 для деревьев кратчайших путей. Чтобы доказать первое свойство, необходимо показать, что V_π — множество вершин, достижимых из истока s. По определению вес кратчайшего пути $\delta(s, v)$ выражается конечной величиной тогда и только тогда, когда вершина v достижима из истока s, поэтому из истока s достижимы только те вершины, которым соответствуют конечные значения d. Однако атрибуту $v.d$ вершины $v \in V - \{s\}$ конечное

значение присваивается тогда и только тогда, когда $v.\pi \neq$ NIL. Таким образом, в множество V_π входят только те вершины, которые достижимы из истока s.

Второе свойство следует непосредственно из леммы 24.16.

Поэтому остается доказать справедливость последнего свойства для дерева кратчайших путей: для каждой вершины $v \in V_\pi$ единственный простой путь $s \overset{p}{\leadsto} v$ в графе G_π — это кратчайший путь в графе G из истока s в вершину v. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$ и $v_k = v$. Для $i = 1, 2, \ldots, k$ выполняются соотношения $v_i.d = \delta(s, v_i)$ и $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$, из чего следует, что $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. В результате суммирования весов вдоль пути p получаем

$$\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\
&= \delta(s, v_k) - \delta(s, v_0) && \text{(в силу “телескопичности” суммы)} \\
&= \delta(s, v_k) && \text{(поскольку } \delta(s, v_0) = \delta(s, s) = 0\text{)} .
\end{aligned}$$

Таким образом, $w(p) \leq \delta(s, v_k)$. Поскольку $\delta(s, v_k)$ является нижней границей веса любого пути из s в v_k, мы заключаем, что $w(p) = \delta(s, v_k)$, и, таким образом, p является кратчайшим путем из s в $v = v_k$. ∎

Упражнения

24.5.1

Для ориентированного графа, изображенного на рис. 24.2, приведите пример двух деревьев кратчайших путей, отличных от показанных.

24.5.2

Приведите пример взвешенного ориентированного графа $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ и истоком s, для которого выполняется следующее свойство: для каждого ребра $(u, v) \in E$ существует дерево кратчайших путей с корнем s, содержащее ребро (u, v), и другое дерево кратчайших путей с корнем s, в котором это ребро отсутствует.

24.5.3

Усовершенствуйте доказательство леммы 24.10, чтобы она охватывала случаи, когда веса кратчайших путей равны ∞ и $-\infty$.

24.5.4

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с истоком s, инициализированный процедурой INITIALIZE-SINGLE-SOURCE(G, s). Докажите, что если в результате выполнения последовательности шагов ослабления

атрибуту $s.\pi$ присваивается значение, отличное от NIL, то граф G содержит цикл с отрицательным весом.

24.5.5

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф, не содержащий ребер с отрицательным весом, и пусть $s \in V$ — исток. Предположим, что в качестве $v.\pi$ может выступать предшественник вершины v на *любом* кратчайшем пути к вершине v из истока s, если вершина $v \in V - \{s\}$ достижима из вершины s; в противном случае $v.\pi$ принимает значение NIL. Приведите пример такого графа G и значений, которые следует присвоить атрибутам π, чтобы в графе G_π образовался цикл. (Согласно лемме 24.16 такое присвоение не может быть получено в результате последовательности шагов ослаблений.)

24.5.6

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$, не содержащий циклов с отрицательным весом. Пусть $s \in V$ — исток, а граф G инициализируется процедурой INITIALIZE-SINGLE- SOURCE(G, s). Докажите, что для каждой вершины $v \in V_\pi$ в графе G_π существует путь из вершины s в вершину v и что это свойство поддерживается как инвариант в ходе произвольной последовательности ослаблений.

24.5.7

Пусть $G = (V, E)$ представляет собой взвешенный ориентированный граф, не содержащий циклов с отрицательным весом. Пусть также $s \in V$ — исток, а граф G инициализируется процедурой INITIALIZE-SINGLE-SOURCE(G, s). Докажите, что существует такая последовательность $|V| - 1$ шагов ослабления, что для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$.

24.5.8

Пусть G — произвольный взвешенный ориентированный граф, который содержит цикл с отрицательным весом, достижимый из истока s. Покажите, что всегда можно построить такую бесконечную последовательность этапов ослабления ребер графа G, что каждое ослабление будет приводить к изменению оценки кратчайшего пути.

Задачи

24.1. Улучшение Йена алгоритма Беллмана–Форда

Предположим, что в каждом проходе алгоритма Беллмана–Форда ослабления ребер упорядочены следующим образом. Перед первым проходом вершины входного графа $G = (V, E)$ выстраиваются в произвольном линейном порядке $v_1, v_2, \ldots, v_{|V|}$. Затем множество ребер E разбивается на множества $E_f \cup E_b$, где $E_f = \{(v_i, v_j) \in E : i < j\}$ и $E_b = \{(v_i, v_j) \in E : i > j\}$. (Предполагается, что

граф G не содержит петель, так что каждое ребро принадлежит либо множеству E_f, либо множеству E_b.) Определим графы $G_f = (V, E_f)$ и $G_b = (V, E_b)$.

а. Докажите, что G_f представляет собой ациклический граф с топологической сортировкой $\langle v_1, v_2, \ldots, v_{|V|} \rangle$, а G_b — ациклический граф с топологической сортировкой $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

Предположим, что каждый проход алгоритма Беллмана–Форда реализован следующим образом. Мы посещаем каждую вершину в порядке $v_1, v_2, \ldots, v_{|V|}$, ослабляя ребра E_f, покидающие эту вершину. Затем мы посещаем каждую вершину в порядке $v_{|V|}, v_{|V|-1}, \ldots, v_1$, ослабляя ребра E_b, покидающие эту вершину.

б. Докажите, что если при такой схеме граф G не содержит циклов с отрицательным весом, достижимых из источника s, то после всего лишь $\lceil |V|/2 \rceil$ проходов по ребрам для всех вершин $v \in V$ выполняется равенство $v.d = \delta(s, v)$.

в. Улучшает ли эта схема асимптотическое время работы алгоритма Беллмана–Форда?

24.2. Вложенные боксы

Говорят, что d-мерный бокс с размерами $(x_1, x_2, \ldots, x_d)$ ***вкладывается*** (nests) в другой бокс с размерами $(y_1, y_2, \ldots, y_d)$, если существует такая перестановка π индексов $\{1, 2, \ldots, d\}$, что выполняются неравенства $x_{\pi(1)} < y_1$, $x_{\pi(2)} < y_2$, ..., $x_{\pi(d)} < y_d$.

а. Докажите, что вложение обладает свойством транзитивности.

б. Разработайте эффективный метод определения, вкладывается ли один d-мерный бокс в другой.

в. Предположим, что задано множество, состоящее из n штук d-мерных боксов $\{B_1, B_2, \ldots, B_n\}$. Разработайте эффективный алгоритм, позволяющий найти самую длинную последовательность боксов $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$, такую, что для $j = 1, 2, \ldots, k-1$ бокс B_{i_j} вкладывается в бокс $B_{i_{j+1}}$. Выразите время работы этого алгоритма через n и d.

24.3. Арбитражные операции

Арбитражные операции (arbitrage) — это использование различий в текущем курсе валют для преобразования единицы валюты в большее количество единиц этой же валюты. Например, предположим, что за один американский доллар можно купить 49 индийских рупий, за одну индийскую рупию — 2 японские иены, а за одну японскую иену — 0.0107 американского доллара. В этом случае в результате ряда операций обмена торговец может начать с одного американского доллара и купить $49 \times 2 \times 0.0107 = 1.0486$ американского доллара, получив, таким образом, доход в размере 4.86%.

Предположим, что даны n валют $c_1, c_2, \ldots, c_n$ и таблица R размером $n \times n$, содержащая обменные курсы, т.е. за одну единицу валюты c_i можно купить $R[i, j]$ единиц валюты c_j.

а. Разработайте эффективный алгоритм, позволяющий определить, существует ли последовательность валют $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$, такая, что

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

Проанализируйте время работы этого алгоритма.

б. Разработайте эффективный алгоритм поиска такой последовательности, если она существует. Проанализируйте время работы этого алгоритма.

24.4. Алгоритм Габова масштабирования кратчайших путей из одной вершины

Алгоритм ***масштабирования*** (scaling) решает задачу, сначала рассматривая только старший бит каждой соответствующей входной величины (такой, как вес ребра). Затем начальное решение улучшается за счет рассмотрения двух старших битов. После этого последовательно рассматривается все большее число (старших) битов, в результате чего каждый раз решение улучшается до тех пор, пока не будут рассмотрены все биты и не будет найдено правильное решение.

В этой задаче исследуется алгоритм вычисления кратчайших путей из одной вершины путем масштабирования весов ребер. Задан ориентированный граф $G = (V, E)$, веса ребер в котором выражаются неотрицательными целыми величинами w. Введем величину $W = \max_{(u,v) \in E} \{w(u, v)\}$. Наша цель – разработать алгоритм, время работы которого составляет $O(E \lg W)$. Предполагается, что все вершины достижимы из истока.

В алгоритме последовательно по одному обрабатываются биты, образующие бинарное представление весов ребер, от самого старшего к самому младшему. В частности, пусть $k = \lceil \lg(W + 1) \rceil$ — количество битов в бинарном представлении величины W и пусть $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$ для $i = 1, 2, \ldots, k$. Другими словами, $w_i(u, v)$ – "масштабированная" версия $w(u, v)$, полученная из i старших битов этой величины. (Таким образом, для всех ребер $(u, v) \in E$ выполняется $w_k(u, v) = w(u, v)$.) Например, если $k = 5$ и $w(u, v) = 25$ (бинарное представление этого значения имеет вид $\langle 11001 \rangle$), то $w_3(u, v) = \langle 110 \rangle = 6$. Другой пример с $k = 5$ – если $w(u, v) = \langle 00100 \rangle = 4$, то $w_3(u, v) = \langle 001 \rangle = 1$. Определим величину $\delta_i(u, v)$ как вес кратчайшего пути из вершины u в вершину v, вычисленный с помощью весовой функции w_i. Таким образом, $\delta_k(u, v) = \delta(u, v)$ для всех вершин $u, v \in V$. Для заданного истока s в алгоритме масштабирования для всех вершин $v \in V$ сначала вычисляются веса кратчайших путей $\delta_1(s, v)$, затем для всех вершин вычисляются величины $\delta_2(s, v)$, и так до тех пор, пока для всех вершин $v \in V$ не будут вычислены величины $\delta_k(s, v)$. Предполагается, что $|E| \geq |V| - 1$. Как вы увидите, вычисление величины δ_i на основании δ_{i-1} требует времени $O(E)$, так что время работы всего алгоритма – $O(kE) = O(E \lg W)$.

а. Предположим, что для всех вершин $v \in V$ выполняется неравенство $\delta(s, v) \leq |E|$. Покажите, что величину $\delta(s, v)$ для всех вершин можно вычислить за время $O(E)$.

б. Покажите, что можно вычислить $\delta_1(s, v)$ для всех $v \in V$ за время $O(E)$.

Теперь рассмотрим вопрос вычисления δ_i из δ_{i-1}.

в. Докажите, что для $i = 2, 3, \ldots, k$ мы имеем либо $w_i(u, v) = 2w_{i-1}(u, v)$, либо $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Затем докажите, что

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

для всех $v \in V$.

г. Определим для $i = 2, 3, \ldots, k$ и всех $(u, v) \in E$

$$\widehat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) .$$

Докажите, что для $i = 2, 3, \ldots, k$ и всех $u, v \in V$ "перевешенное" значение $\widehat{w}_i(u, v)$ ребра (u, v) представляет собой неотрицательное целое число.

д. Теперь определим $\widehat{\delta}_i(s, v)$ как вес кратчайшего пути из s в v с использованием весовой функции $\widehat{w}_i$. Докажите, что для $i = 2, 3, \ldots, k$ и всех $v \in V$

$$\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

и что $\widehat{\delta}_i(s, v) \leq |E|$.

е. Покажите, как вычислить $\delta_i(s, v)$ из $\delta_{i-1}(s, v)$ для всех $v \in V$ за время $O(E)$, и сделайте вывод о том, что можно вычислить $\delta(s, v)$ для всех $v \in V$ за время $O(E \lg W)$.

24.5. Алгоритм Карпа для поиска цикла с минимальным средним весом

Пусть $G = (V, E)$ представляет собой ориентированный граф с весовой функцией $w : E \to \mathbb{R}$ и пусть $n = |V|$. Определим ***средний вес*** (mean weight) цикла $c = \langle e_1, e_2, \ldots, e_k \rangle$, состоящего из ребер множества E, как

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i) .$$

Пусть $\mu^* = \min_c \mu(c)$, где c охватывает все ориентированные циклы графа G. Цикл c, для которого выполняется равенство $\mu(c) = \mu^*$, называется ***циклом с минимальным средним весом*** (minimum mean-weight cycle). В этой задаче исследуется эффективный алгоритм вычисления величины μ^*.

Без потери общности предположим, что каждая вершина $v \in V$ достижима из истока $s \in V$. Пусть $\delta(s, v)$ — вес кратчайшего пути из истока s в вершину v, а $\delta_k(s, v)$ — вес кратчайшего пути из истока s в вершину v, содержащего *ровно* k ребер. Если такого пути не существует, то $\delta_k(s, v) = \infty$.

а. Покажите, что если $\mu^* = 0$, то граф G не содержит циклов с отрицательным весом и $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ для всех вершин $v \in V$.

б. Покажите, что если $\mu^* = 0$, то

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \ge 0$$

для всех вершин $v \in V$. (*Указание:* воспользуйтесь обоими свойствами из п. (а).)

в. Пусть c — цикл с нулевым весом, а u и v — две произвольные вершины в этом цикле. Предположим, что $\mu^* = 0$ и что вес пути из вершины u в вершину v вдоль цикла равен x. Докажите, что $\delta(s,v) = \delta(s,u) + x$. (*Указание:* вес простого пути из вершины v в вершину u вдоль цикла равен $-x$.)

г. Покажите, что если $\mu^* = 0$, то в каждом цикле с минимальным средним весом существует вершина v, такая, что

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0 .$$

(*Указание:* покажите, как можно расширить кратчайший путь в каждую вершину, принадлежащую циклу с минимальным средним весом, вдоль цикла, чтобы получить путь к следующей вершине цикла.)

д. Покажите, что если $\mu^* = 0$, то

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0 .$$

е. Покажите, что если к весу каждого ребра графа G добавить константу t, то величина μ^* увеличится на t. Покажите с помощью этого факта справедливость соотношения

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} .$$

ж. Разработайте алгоритм, позволяющий вычислить величину μ^* за время $O(VE)$.

24.6. Битонические кратчайшие пути

Последовательность называется ***битонической*** (bitonic), если она монотонно возрастает, а затем монотонно убывает, или если путем циклического сдвига ее можно привести к такому виду. Например, битоническими являются последовательности $\langle 1,4,6,8,3,-2 \rangle$, $\langle 9,2,-4,-10,-5 \rangle$ и $\langle 1,2,3,4 \rangle$, но не $\langle 1,3,12,4,2,10 \rangle$. (См. битоническую евклидову задачу о коммивояжере 15.3.)

Предположим, что задан ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ и нужно найти кратчайшие пути из одной вершины s. Имеется также дополнительная информация: для каждой вершины $v \in V$ веса ребер вдоль любого кратчайшего пути из истока s в вершину v образуют битоническую последовательность.

Разработайте наиболее эффективный алгоритм, позволяющий решить эту задачу, и проанализируйте время его работы.

Заключительные замечания

Алгоритм Дейкстры (Dijkstra) [87] был разработан в 1959 году, но в нем не содержалось никаких упоминаний об очереди с приоритетами. Алгоритм Беллмана–Форда основан на отдельных алгоритмах Беллмана (Bellman) [37] и Форда (Ford) [108]. Беллман указал, как кратчайшие пути связаны с разностными ограничениями. Лоулер (Lawler) [223] описал алгоритм с линейным временем работы для поиска кратчайших путей в ориентированном ациклическом графе, который он рассматривает как часть "народного творчества".

Если вес каждого из ребер выражается сравнительно малыми неотрицательными целыми числами, задача о кратчайших путях из одной вершины решается с помощью более эффективных алгоритмов. Последовательность значений, возвращаемых в результате вызовов процедуры EXTRACT-MIN в алгоритме Дейкстры, монотонно возрастает со временем. Как говорилось в заключительных замечаниях к главе 6, в этом случае существует несколько структур данных, позволяющих эффективнее реализовать различные операции над очередями с приоритетами, чем бинарная пирамида или пирамида Фибоначчи. Ахуя (Ahuja), Мельхорн (Mehlhorn), Орлин (Orlin) и Тарджан (Tarjan) [8] предложили для графов с неотрицательными весами ребер алгоритм со временем работы $O(E + V\sqrt{\lg W})$, где W — максимальный вес ребра графа. Наилучшие границы достигнуты Торупом (Thorup) [335], который предложил алгоритм со временем работы $O(E \lg\lg V)$, и Раманом (Raman) [289], алгоритм которого имеет время работы $O\left(E + V \min\left\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\right\}\right)$. Оба алгоритма используют объем памяти, зависящий от размера слова машины, на которой выполняется алгоритм. Хотя объем используемой памяти может оказаться неограниченным в зависимости от размера входных данных, с помощью рандомизированного хеширования его можно снизить до линейно зависящего от размера входных данных.

Для неориентированных графов с целочисленными весами Торуп [334] привел алгоритм со временем работы $O(V + E)$, предназначенный для поиска кратчайших путей из одной вершины. В отличие от алгоритмов, упомянутых в предыдущем абзаце, этот алгоритм — не реализация алгоритма Дейкстры, поскольку последовательность значений, возвращаемых вызовами процедуры EXTRACT-MIN, не является монотонно неубывающей.

Для графов, содержащих ребра с отрицательными весами, алгоритм, предложенный Габовым (Gabow) и Тaржаном [121], имеет время работы $O(\sqrt{V}E\lg(VW))$, а предложенный Гольдбергом (Goldberg) [136] выполняется за время $O(\sqrt{V}E\lg W)$, где $W = \max_{(u,v)\in E}\{|w(u,v)|\}$.

Черкасский (Cherkassky), Гольдберг (Goldberg) и Радзик (Radzik) [63] провели большое количество экспериментов по сравнению различных алгоритмов, предназначенных для поиска кратчайших путей.

Глава 25. Кратчайшие пути между всеми парами вершин

В этой главе рассматривается задача о поиске кратчайших путей между всеми парами вершин графа. Эта задача может возникнуть, например, при составлении таблицы расстояний между всеми парами городов, нанесенных на атлас дорог. Как и в главе 24, в этой задаче задается взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$, отображающей ребра на их веса, выраженные действительными числами. Для каждой пары вершин $u, v \in V$ требуется найти кратчайший (обладающий наименьшим весом) путь из вершины u в вершину v, вес которого определяется как сумма весов входящих в него ребер. Обычно выходные данные представляются в табличной форме: на пересечении строки с индексом u и столбца с индексом v расположен вес кратчайшего пути из вершины u в вершину v.

Задачу о поиске кратчайших путей между всеми парами вершин можно решить, выполнив $|V|$ раз алгоритм поиска кратчайших путей из одной вершины, каждый раз выбирая в качестве истока новую вершину графа. Если веса всех ребер неотрицательные, можно воспользоваться алгоритмом Дейкстры. Если используется реализация неубывающей очереди с приоритетами в виде линейного массива, то время работы такого алгоритма равно $O(V^3 + VE) = O(V^3)$. Если же неубывающая очередь с приоритетами реализована в виде бинарной неубывающей пирамиды, то время работы будет равно $O(VE \lg V)$, что предпочтительнее для разреженных графов. Можно также реализовать неубывающую очередь с приоритетами с помощью пирамиды Фибоначчи; в этом случае время работы алгоритма равно $O(V^2 \lg V + VE)$.

Если в графе могут быть ребра с отрицательным весом, алгоритм Дейкстры неприменим. Вместо него для каждой вершины следует выполнить более медленный алгоритм Беллмана–Форда. Полученное в результате время работы равно $O(V^2 E)$, что для плотных графов можно записать как $O(V^4)$. После чтения этой главы станет понятно, как лучше поступить в том или ином случае. Будет также исследована связь задачи о кратчайших расстояниях между всеми парами вершин с умножением матриц и изучена алгебраическая структура этой задачи.

В отличие от алгоритмов поиска кратчайшего пути из фиксированного истока, в которых предполагается, что представление графа имеет вид списка смежных вершин, в большинстве представленных в этой главе алгоритмов используется представление в виде матрицы смежности. (В алгоритме Джонсона (Johnson) для разреженных графов в разделе 25.3 используются списки смежности.) Для удоб-

ства предполагается, что вершины пронумерованы как $1, 2, \ldots, |V|$, поэтому в роли входных данных выступает матрица W размером $n \times n$, представляющая веса ориентированных ребер (дуг) ориентированного графа $G = (V, E)$ с n вершинами. Другими словами, $W = (w_{ij})$, где

$$w_{ij} = \begin{cases} 0 , & \text{если } i = j , \\ \text{вес дуги } (i,j) , & \text{если } i \neq j \text{ и } (i,j) \in E , \\ \infty , & \text{если } i \neq j \text{ и } (i,j) \notin E . \end{cases} \tag{25.1}$$

Наличие ребер с отрицательным весом допускается, но пока что предполагается, что входной граф не содержит циклов с отрицательным весом.

Выходные данные представленных в этой главе алгоритмов, предназначенных для поиска кратчайших путей между всеми парами вершин, имеют вид матрицы $D = (d_{ij})$ размером $n \times n$, где элемент d_{ij} содержит вес кратчайшего пути из вершины i в вершину j. Другими словами, если обозначить через $\delta(i, j)$ кратчайший путь из вершины i в вершину j (как это было сделано в главе 24), то по завершении работы алгоритма $d_{ij} = \delta(i, j)$.

Чтобы решить задачу о поиске кратчайших путей между всеми парами вершин со входной матрицей смежности, необходимо вычислить не только вес каждого из кратчайших путей, но и ***матрицу предшествования*** (predecessor matrix) $\Pi = (\pi_{ij})$, где величина π_{ij} имеет значение NIL, если $i = j$ или путь из вершины i в вершину j отсутствует; в противном случае π_{ij} — предшественник вершины j на некотором кратчайшем пути из вершины i. Точно так же, как описанный в главе 24 подграф предшествования G_π является деревом кратчайших путей для заданного истока, подграф, индуцированный i-й строкой матрицы Π, должен быть деревом кратчайших путей с корнем i. Определим для каждой вершины $i \in V$ ***подграф предшествования*** (predecessor subgraph) графа G для вершины i как граф $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, где

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

и

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\} .$$

Если $G_{\pi,i}$ является деревом кратчайших путей, то приведенная ниже процедура, представляющая собой модифицированную версию описанной в главе 22 процедуры PRINT-PATH, выводит кратчайший путь из вершины i в вершину j.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i == j
    print i
elseif π_ij == NIL
    print "Пути из" i "в" j "не существует"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    print j
```

Чтобы подчеркнуть важные особенности представленных в этой главе алгоритмов поиска кратчайших путей между всеми парами вершин, создание матриц предшествования и их свойств не будет рассматриваться здесь так же подробно, как в главе 24 в случае подграфа предшествования. Основные моменты предлагается рассмотреть в некоторых упражнениях.

Краткое содержание главы

В разделе 25.1 представлен алгоритм динамического программирования, основанный на операции умножения матриц, который позволяет решить задачу о поиске кратчайших путей между всеми парами вершин. С помощью многократного возведения в квадрат можно сделать так, чтобы время работы этого алгоритма было равно $\Theta(V^3 \lg V)$. В разделе 25.2 приведен другой алгоритм динамического программирования — алгоритм Флойда–Уоршелла (Floyd–Warshall). Время работы этого алгоритма равно $\Theta(V^3)$. В этом же разделе исследуется задача поиска транзитивного замыкания ориентированного графа, связанная с задачей о поиске кратчайших путей между всеми парами вершин. Наконец в разделе 25.3 представлен алгоритм Джонсона. В отличие от других алгоритмов, описанных в этой главе, в алгоритме Джонсона применяется представление графа в виде списка смежных вершин. Этот алгоритм позволяет решить задачу о поиске кратчайших путей между всеми парами вершин за время $O(V^2 \lg V + VE)$, что делает его пригодным для больших разреженных графов.

Перед тем как продолжить, нам нужно принять некоторые соглашения для представлений в виде матрицы смежности. Во-первых, в общем случае предполагается, что входной граф $G = (V, E)$ содержит n вершин, так что $n = |V|$. Во-вторых, будет использоваться соглашение об обозначении матриц прописными буквами, например W, L или D, а их отдельных элементов — строчными буквами с нижними индексами, например w_{ij}, l_{ij} или d_{ij}. Возле некоторых матриц будут приведены заключенные в скобки верхние индексы, указывающие на количество выполненных итераций, например $L^{(m)} = \left(l_{ij}^{(m)}\right)$ или $D^{(m)} = \left(d_{ij}^{(m)}\right)$. Наконец для заданной матрицы A размером $n \times n$ предполагается, что значение n хранится в атрибуте $A.rows$.

25.1. Задача о кратчайших путях и умножение матриц

В этом разделе представлен алгоритм динамического программирования, предназначенный для решения задачи о поиске кратчайших путей между всеми парами вершин в ориентированном графе $G = (V, E)$. В каждом основном цикле динамического программирования будет вызываться операция, очень напоминающая матричное умножение, поэтому такой алгоритм будет напоминать многократное умножение матриц. Начнем с того, что разработаем для решения задачи о кратчайших путях между всеми парами вершин алгоритм со временем работы $\Theta(V^4)$, после чего улучшим этот показатель до величины $\Theta(V^3 \lg V)$.

Перед тем как продолжить, кратко напомним описанные в главе 15 этапы разработки алгоритма динамического программирования.

1. Описание структуры оптимального решения.
2. Рекурсивное определение значения оптимального решения.
3. Вычисление значения оптимального решения восходящим методом.

(Этап 4, состоящий в составлении оптимального решения на основе полученной информации, рассматривается в упражнениях.)

Структура кратчайшего пути

Начнем с того, что охарактеризуем структуру оптимального решения. Для задачи о кратчайших путях между всеми парами вершин графа $G = (V, E)$ доказано (лемма 24.1), что все подпути кратчайшего пути — также кратчайшие пути. Предположим, что граф представлен матрицей смежности $W = (w_{ij})$. Рассмотрим кратчайший путь p из вершины i в вершину j и предположим, что этот путь содержит не более m ребер. Если циклы с отрицательным весом отсутствуют, то значение m конечно. Если $i = j$, то вес пути p равен 0, а ребра в нем отсутствуют. Если же вершины i и j различаются, то путь p раскладывается на $i \overset{p'}{\leadsto} k \to j$, где путь p' содержит не более $m-1$ ребер. Согласно лемме 24.1 p' — кратчайший путь из вершины i в вершину k, поэтому выполняется равенство $\delta(i,j) = \delta(i,k) + w_{kj}$.

Рекурсивное решение задачи о кратчайших путях между всеми парами вершин

Пусть теперь $l_{ij}^{(m)}$ — минимальный вес любого пути из вершины i в вершину j, содержащий не более m ребер. Если $m = 0$, то кратчайший путь из вершины i в вершину j существует тогда и только тогда, когда $i = j$. Таким образом,

$$l_{ij}^{(0)} = \begin{cases} 0\,, & \text{если } i = j\,, \\ \infty\,, & \text{если } i \neq j\,. \end{cases}$$

Для $m \geq 1$ величина $l_{ij}^{(m)}$ вычисляется как минимум двух величин. Первая из них — $l_{ij}^{(m-1)}$ (вес кратчайшего пути из вершины i в вершину j, состоящего не более чем из $m-1$ ребер), а вторая — минимальный вес произвольного пути из вершины i в вершину j, который состоит не более чем из m ребер. Этот минимальный вес получается в результате рассмотрения всех возможных предшественников k вершины j. Таким образом, мы можем рекурсивно определить

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\,. \end{aligned} \tag{25.2}$$

Последнее равенство следует из того, что $w_{jj} = 0$ для всех j.

Чему равен фактический вес каждого из кратчайших путей $\delta(i, j)$? Если граф не содержит циклов с отрицательным весом, то для каждой пары вершин i и j, для которых справедливо неравенство $\delta(i, j) < \infty$, существует кратчайший путь из вершины i в вершину j, который является простым и, следовательно, содержит не более $n - 1$ ребер. Путь из вершины i в вершину j, содержащий более $n - 1$ ребер, не может иметь меньший вес, чем кратчайший путь из вершины i в вершину j. Поэтому фактический вес каждого из кратчайших путей определяется равенствами

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots . \tag{25.3}$$

Вычисление весов кратчайших путей в восходящем порядке

Используя в качестве входной матрицу $W = (w_{ij})$, вычислим ряд матриц $L^{(1)}$, $L^{(2)}, \ldots, L^{(n-1)}$, где для $m = 1, 2, \ldots, n - 1$ имеем $L^{(m)} = \left(l_{ij}^{(m)}\right)$. Конечная матрица $L^{(n-1)}$ содержит фактический вес каждого из кратчайших путей. Заметим, что для всех вершин $i, j \in V$ выполняется равенство $l_{ij}^{(1)} = w_{ij}$, так что $L^{(1)} = W$.

Сердцем алгоритма является приведенная ниже процедура, которая на основе заданных матриц $L^{(m-1)}$ и W вычисляет и возвращает матрицу $L^{(m)}$. Другими словами, она расширяет вычисленные на текущий момент кратчайшие пути, добавляя в них еще по одному ребру.

```
EXTEND-SHORTEST-PATHS(L, W)
n = L.rows
Пусть L' = (l'_ij) — новая матрица размером n × n
for i = 1 to n
    for j = 1 to n
        l'_ij = ∞
        for k = 1 to n
            l'_ij = min(l'_ij, l_ik + w_kj)
return L'
```

В этой процедуре вычисляется матрица $L' = (l'_{ij})$, которая и возвращается процедурой по завершении работы. Вычисления осуществляются на основе уравнения (25.2) для всех пар индексов i и j; при этом в качестве $L^{(m-1)}$ используется матрица L, а в качестве $L^{(m)}$ — матрица L'. (В псевдокоде верхние индексы не используются, чтобы входные и выходные матрицы процедуры не зависели от m.) Из-за наличия трех вложенных циклов **for** время работы алгоритма равно $\Theta(n^3)$.

Теперь становится понятной связь с умножением матриц. Предположим, требуется вычислить матричное произведение $C = A \cdot B$, где A и B — матрицы размером $n \times n$. Тогда для $i, j = 1, 2, \ldots, n$ мы вычисляем

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} . \tag{25.4}$$

Заметим, что если выполнить замены

$$\begin{aligned} l^{(m-1)} &\to a\ , \\ w &\to b\ , \\ l^{(m)} &\to c\ , \\ \min &\to +\ , \\ + &\to \cdot \end{aligned}$$

в уравнении (25.2), то получится уравнение (25.4). Таким образом, если в процедуре EXTEND-SHORTEST-PATHS провести соответствующие изменения, а также заменить значение ∞ (исходное значение для операции вычисления минимума) значением 0 (исходное значение для вычисления суммы), получится процедура для непосредственного перемножения матриц со временем выполнения $\Theta(n^3)$, которую мы уже видели в разделе 4.2.

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
Пусть C — новая матрица размером n × n
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Возвращаясь к задаче о кратчайших путях между всеми парами вершин, вычислим веса кратчайших путей путем поэтапного расширения путей ребро за ребром. Обозначив через $A \cdot B$ матричное “произведение”, которое возвращается процедурой EXTEND-SHORTEST-PATHS(A, B), вычислим последовательность $n-1$ матриц

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &&= W\ , \\ L^{(2)} &= L^{(1)} \cdot W &&= W^2\ , \\ L^{(3)} &= L^{(2)} \cdot W &&= W^3\ , \\ &\ \ \vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}\ . \end{aligned}$$

Как было показано ранее, матрица $L^{(n-1)} = W^{n-1}$ содержит веса кратчайших путей. В приведенной ниже процедуре эта последовательность вычисляется за время $\Theta(n^4)$.

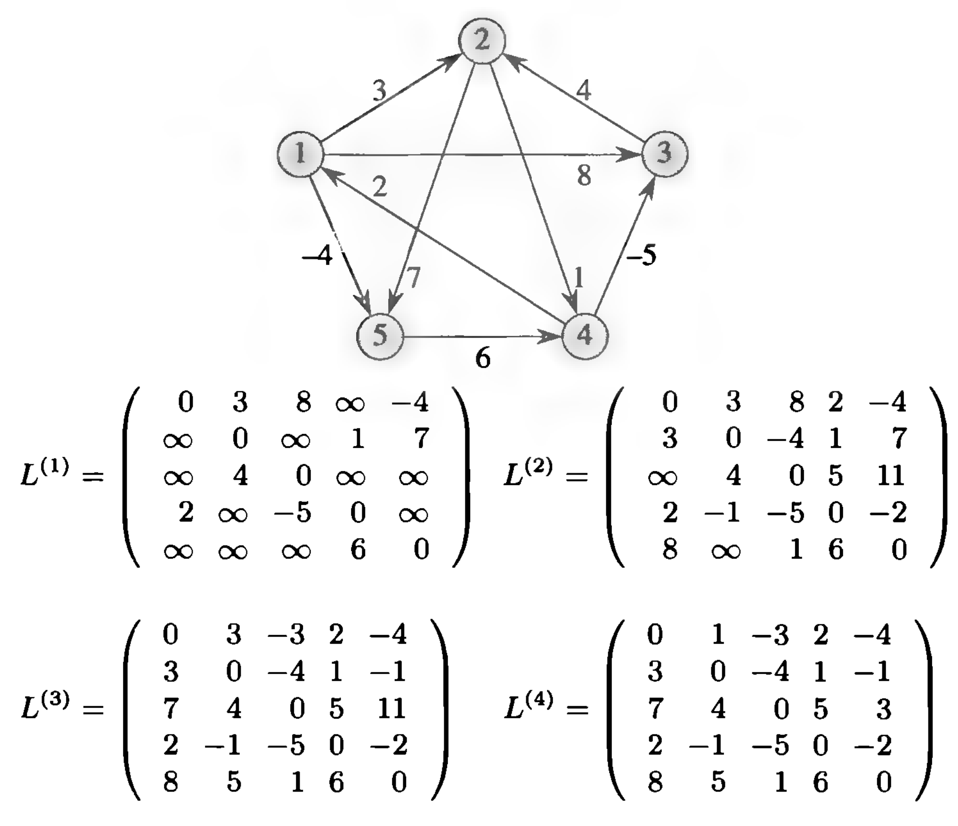


Рис. 25.1. Ориентированный граф и последовательность матриц $L^{(m)}$, вычисляемых процедурой SLOW-ALL-PAIRS-SHORTEST-PATHS. Можно легко убедиться в том, что величина $L^{(5)} = L^{(4)} \cdot W$ равна $L^{(4)}$, а следовательно, для всех $m \geq 4$ выполняется равенство $L^{(m)} = L^{(4)}$.

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
    Пусть $L^{(m)}$ — новая матрица размером $n \times n$
    $L^{(m)} =$ EXTEND-SHORTEST-PATHS($L^{(m-1)}, W$)
return $L^{(n-1)}$

На рис. 25.1 приведены граф и матрицы $L^{(m)}$, вычисленные процедурой SLOW-ALL-PAIRS-SHORTEST-PATHS.

Улучшение времени работы

Однако наша цель состоит не в том, чтобы вычислить *все* матрицы $L^{(m)}$: нам нужна только одна матрица — $L^{(n-1)}$. Напомним, что если циклы с отрицательным весом отсутствуют, из уравнения (25.3) вытекает равенство $L^{(m)} = L^{(n-1}$ для всех целых $m \geq n - 1$. Матричное умножение, определенное процедурой EXTEND-SHORTEST-PATHS, как и обычное матричное умножение, является ассоциативным (упр. 25.1.4). Таким образом, матрицу $L^{(n-1)}$ можно получить путем

вычисления $\lceil \lg(n-1) \rceil$ матричных умножений в последовательности

$$\begin{aligned}
L^{(1)} &= W , \\
L^{(2)} &= W^2 &&= W \cdot W , \\
L^{(4)} &= W^4 &&= W^2 \cdot W^2 \\
L^{(8)} &= W^8 &&= W^4 \cdot W^4 , \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}} .
\end{aligned}$$

Поскольку $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, последнее произведение $L^{(2^{\lceil \lg(n-1) \rceil})}$ равно $L^{(n-1)}$.

В приведенной ниже процедуре указанная последовательность матриц вычисляется методом ***многократного возведения в квадрат*** (repeated squaring).

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n − 1
5      Пусть L^(2m) — новая матрица размером n × n
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

В каждой итерации цикла **while** в строках 4–7, начиная с $m = 1$ вычисляется матрица $L^{(2m)} = \left(L^{(m)}\right)^2$. В конце каждой итерации значение m удваивается. В последней итерации матрица $L^{(n-1)}$ вычисляется путем фактического вычисления матрицы $L^{(2m)}$ для некоторого значения $n - 1 \leq 2m \leq 2n - 2$. Согласно уравнению (25.3) $L^{(2m)} = L^{(n-1)}$. Далее выполняется проверка в строке 4, значение m удваивается, после чего выполняется условие $m \geq n - 1$, так что условие цикла оказывается невыполненным, и процедура возвращает последнюю вычисленную матрицу.

Поскольку каждое из $\lceil \lg(n-1) \rceil$ матричных произведений требует времени $\Theta(n^3)$, процедура FASTER-ALL-PAIRS-SHORTEST-PATHS выполняется за время $\Theta(n^3 \lg n)$. Заметим, что код процедуры довольно компактен. Он не содержит сложных структур данных, поэтому константа, скрытая в Θ-обозначении, невелика.

Упражнения

25.1.1

Выполните процедуру SLOW-ALL-PAIRS-SHORTEST-PATHS над взвешенным ориентированным графом, показанным на рис. 25.2, и запишите промежуточные матрицы, которые получаются в каждой итерации цикла. Затем проделайте то же самое для процедуры FASTER-ALL-PAIRS-SHORTEST-PATHS.

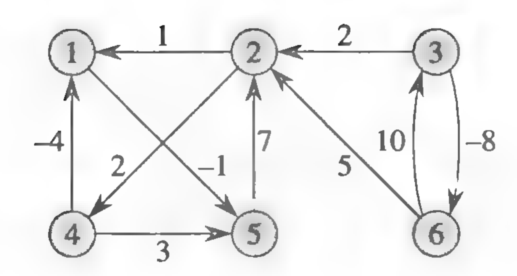


Рис. 25.2. Взвешенный ориентированный граф к упр. 25.1.1, 25.2.1 и 25.3.1.

25.1.2

Почему требуется, чтобы при всех $1 \leq i \leq n$ выполнялось равенство $w_{ii} = 0$?

25.1.3

Чему в операции обычного матричного умножения соответствует матрица

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix},$$

используемая в алгоритмах поиска кратчайших путей?

25.1.4

Покажите, что матричное умножение, определенное в процедуре EXTEND-SHORTEST-PATHS, обладает свойством ассоциативности.

25.1.5

Покажите, как выразить задачу о кратчайшем пути из единого истока в виде произведения матриц и вектора. Опишите, как вычисление этого произведения соответствует алгоритму, аналогичного алгоритму Беллмана–Форда (см. раздел 24.1).

25.1.6

Предположим, что в алгоритмах, о которых идет речь в этом разделе, нужно также найти вершины, принадлежащие кратчайшим путям. Покажите, как за время $O(n^3)$ вычислить матрицу предшествования Π на основе известной матрицы весов кратчайших путей L.

25.1.7

Вершины, принадлежащие кратчайшим путям, можно вычислить одновременно с весом каждого из кратчайших путей. Определим $\pi_{ij}^{(m)}$ как предшественник вершины j на произвольном пути с минимальным весом, который соединяет вершины i и j и содержит не более m ребер. Модифицируйте процедуры EXTEND-SHORTEST-PATHS и SLOW-ALL-PAIRS-SHORTEST-PATHS таким образом, чтобы они позволяли вычислять матрицы $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ наряду с матрицами $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.

25.1.8

В процедуре FASTER-ALL-PAIRS-SHORTEST-PATHS в том виде, в котором она представлена, требуется хранить $\lceil \lg(n-1) \rceil$ матриц, содержащих по n^2 элементов, для чего требуется общий объем памяти, равный $\Theta(n^2 \lg n)$. Модифицируйте данную процедуру таким образом, чтобы ей требовалось только $\Theta(n^2)$ памяти, используя для работы только две матрицы размером $n \times n$.

25.1.9

Модифицируйте процедуру FASTER-ALL-PAIRS-SHORTEST-PATHS таким образом, чтобы она была способна выявлять наличие в графе циклов с отрицательным весом.

25.1.10

Разработайте эффективный алгоритм, позволяющий находить в графе количество ребер минимального по длине цикла с отрицательным весом.

25.2. Алгоритм Флойда–Уоршелла

В этом разделе задача о поиске кратчайших путей между всеми парами вершин в ориентированном графе $G = (V, E)$ будет решаться с помощью различных подходов динамического программирования. Время работы полученного в результате алгоритма, известного как ***алгоритм Флойда–Уоршелла*** (Floyd–Warshall algorithm), равно $\Theta(V^3)$. Как и ранее, наличие ребер с отрицательным весом допускается, но предполагается, что циклы с отрицательным весом отсутствуют. Как и в разделе 25.1, нами будет пройден весь процесс разработки алгоритма в стиле динамического программирования. После исследования полученного в результате алгоритма будет представлен аналогичный метод для поиска транзитивного замыкания ориентированного графа.

Структура кратчайшего пути

В алгоритме Флойда–Уоршелла используется характеристика структуры кратчайшего пути, отличная от рассмотренной в разделе 25.1. В этом алгоритме рассматриваются промежуточные вершины кратчайшего пути. ***Промежуточной*** (intermediate) вершиной простого пути $p = \langle v_1, v_2, \ldots, v_l \rangle$ является любая вершина p, отличная от v_1 и v_l, т.е. любая вершина множества $\{v_2, v_3, \ldots, v_{l-1}\}$.

Алгоритм Флойда–Уоршелла основан на следующем наблюдении. Предположим, что граф G состоит из вершин $V = \{1, 2, \ldots, n\}$. Рассмотрим подмножество вершин $\{1, 2, \ldots, k\}$ для некоторого k. Для произвольной пары вершин $i, j \in V$ рассмотрим все пути из вершины i в вершину j, все промежуточные вершины которых выбраны из множества $\{1, 2, \ldots, k\}$. Пусть среди этих путей p — путь с минимальным весом (этот путь простой). В алгоритме Флойда–Уоршелла используется взаимосвязь между путем p и кратчайшими путями из вершины i в вершину j, все промежуточные вершины которых принадлежат множе-

Все промежуточные вершины из $\{1, 2, \ldots, k-1\}$ Все промежуточные вершины из $\{1, 2, \ldots, k-1\}$

p_1 k p_2 i j

p: Все промежуточные вершины из $\{1, 2, \ldots, k\}$

Рис. 25.3. Путь p является кратчайшим путем из вершины i в вершину j, а k — промежуточная вершина p с наибольшим номером. Все вершины пути p_1, являющегося частью пути p из вершины i в вершину k, принадлежат множеству $\{1, 2, \ldots, k-1\}$. То же самое относится и к пути p_2 из вершины k в вершину j.

ству $\{1, 2, \ldots, k-1\}$. Эта взаимосвязь зависит от того, является ли вершина k промежуточной на пути p.

- Если k не является промежуточной вершиной пути p, то все промежуточные вершины этого пути принадлежат множеству $\{1, 2, \ldots, k-1\}$. Таким образом, кратчайший путь из вершины i в вершину j со всеми промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$ одновременно является кратчайшим путем из вершины i в вершину j со всеми промежуточными вершинами из множества $\{1, 2, \ldots, k\}$.
- Если k является промежуточной вершиной пути p, то этот путь, как видно из рис. 25.3, можно разбить следующим образом: $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$. Согласно лемме 24.1 p_1 является кратчайшим путем из вершины i в вершину k, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$. Поскольку k не является промежуточной вершиной пути p_1, все промежуточные вершины этого пути принадлежит множеству $\{1, 2, \ldots, k-1\}$. Следовательно, p_1 является кратчайшим путем от i до k, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$. Аналогично p_2 — кратчайший путь из вершины k в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$.

Рекурсивное решение задачи о кратчайших путях между всеми парами вершин

Определим на основе сделанных выше наблюдений рекурсивную формулировку оценок кратчайших путей, отличную от той, которая использовалась в разделе 25.1. Пусть $d_{ij}^{(k)}$ — вес кратчайшего пути из вершины i в вершину j, для которого все промежуточные вершины принадлежат множеству $\{1, 2, \ldots, k\}$. Если $k = 0$, то путь из вершины i в вершину j, в котором отсутствуют промежуточные вершины с номером, бóльшим нуля, не содержит промежуточных вершин вообще. Такой путь содержит не более одного ребра, следовательно, $d_{ij}^{(0)} = w_{ij}$. Рекурсивное определение, которое соответствует приведенному выше описанию,

задается соотношением

$$d_{ij}^{(k)} = \begin{cases} w_{ij} \ , & \text{если } k = 0 \ , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \ , & \text{если } k \geq 1 \ . \end{cases} \tag{25.5}$$

Поскольку все промежуточные вершины произвольного пути принадлежат множеству $\{1, 2, \ldots, n\}$, матрица $D^{(n)} = \left(d_{ij}^{(n)}\right)$ дает окончательный ответ: $d_{ij}^{(n)} = \delta(i, j)$ для всех пар вершин $i, j \in V$.

Вычисление весов кратчайших путей в восходящем порядке

На основе рекуррентного соотношения (25.5) можно создать приведенную ниже процедуру, предназначенную для вычисления величин $d_{ij}^{(k)}$ в порядке возрастания k. В качестве входных данных выступает матрица W размером $n \times n$, определенная, как в уравнении (25.1). Процедура возвращает матрицу $D^{(n)}$, содержащую веса кратчайших путей.

FLOYD-WARSHALL(W)

1 $n = W.rows$
2 $D^{(0)} = W$
3 **for** $k = 1$ **to** n
4   Пусть $D^{(k)} = \left(d_{ij}^{(k)}\right)$ — новая матрица размером $n \times n$
5   **for** $i = 1$ **to** n
6   **for** $j = 1$ **to** n
7             $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8 **return** $D^{(n)}$

На рис. 25.4 показаны матрицы $D^{(k)}$, вычисленные алгоритмом Флойда–Уоршелла для графа, изображенного на рис. 25.1.

Время работы алгоритма Флойда–Уоршелла определяется трижды вложенными друг в друга циклами **for** в строках 3–7. Поскольку для каждого выполнения строки 7 требуется время $O(1)$, время работы всего алгоритма составляет $\Theta(n^3)$. Код этого алгоритма так же компактен, как и код алгоритма из раздела 25.1. Он не содержит сложных структур данных, поэтому константа, скрытая в Θ-обозначениях, мала. Таким образом, алгоритм Флойда–Уоршелла имеет практическую ценность даже для входных графов среднего размера.

Построение кратчайшего пути

Существует множество различных методов, позволяющих строить кратчайшие пути в алгоритме Флойда–Уоршелла. Один из них — вычисление матрицы D, содержащей веса кратчайших путей, с последующим конструированием на ее основе матрицы предшествования Π. Этот метод можно реализовать таким образом, чтобы время его выполнения было равно $O(n^3)$ (упр. 25.1.6). Если задана мат-

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Рис. 25.4. Последовательность матриц $D^{(k)}$ и $\Pi^{(k)}$, вычисленных алгоритмом Флойда–Уоршелла для графа, изображенного на рис. 25.1.

рица предшествования Π, то вывести вершины на указанном кратчайшем пути можно с помощью процедуры PRINT-ALL-PAIRS-SHORTEST-PATH.

Матрицу предшествования Π можно так же вычислить "на лету", в процессе вычисления в алгоритме Флойда–Уоршелла матриц $D^{(k)}$. Точнее говоря, вычисляется последовательность матриц $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, где $\Pi = \Pi^{(n)}$, а элемент $\pi_{ij}^{(k)}$ определяется как предшественник вершины j на кратчайшем пути из вершины i, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$.

Можно дать рекурсивное определение величины $\pi_{ij}^{(k)}$. Когда $k = 0$, кратчайший путь из вершины i в вершину j не содержит промежуточных вершин. Таким

образом,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} , & \text{если } i = j \text{ или } w_{ij} = \infty , \\ i , & \text{если } i \neq j \text{ и } w_{ij} < \infty . \end{cases} \quad (25.6)$$

Если при $k \geq 1$ получаем путь $i \rightsquigarrow k \rightsquigarrow j$, где $k \neq j$, то выбранный предшественник вершины j совпадает с выбранным предшественником этой же вершины на кратчайшем пути из вершины k, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$. В противном случае выбирается тот же предшественник вершины j, который был выбран на кратчайшем пути из вершины i, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$. Выражаясь формально, при $k \geq 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} , & \text{если } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} , & \text{если } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases} \quad (25.7)$$

Вопрос о том, как включить вычисление матрицы $\Pi^{(k)}$ в процедуру FLOYD-WARSHALL, предлагается рассмотреть в упр. 25.2.3 самостоятельно. На рис. 25.4 показана последовательность матриц $\Pi^{(k)}$, полученных в результате обработки алгоритмом графа, изображенного на рис. 25.1. В упомянутом упражнении также предлагается выполнить более сложную задачу — доказать, что подграф предшествования $G_{\pi,i}$ является деревом кратчайших путей с корнем i. Еще один способ реконструкции кратчайших путей рассматривается в упр. 25.2.7.

Транзитивное замыкание ориентированного графа

Может возникнуть необходимость установить, существуют ли в заданном ориентированном графе $G = (V, E)$, множество вершин которого — $V = \{1, 2, \ldots, n\}$, пути из вершины i в вершину j для всех возможных пар вершин $i, j \in V$. ***Транзитивное замыкание*** (transitive closure) графа G определяется как граф $G^* = (V, E^*)$, где

$$E^* = \{(i, j) : \text{в графе G имеется путь из вершины } i \text{ в вершину } j\} .$$

Один из способов найти транзитивное замыкание графа в течение времени $\Theta(n^3)$ — присвоить каждому ребру из множества E вес 1 и выполнить алгоритм Флойда–Уоршелла. Если путь из вершины i в вершину j существует, то мы получим $d_{ij} < n$; в противном случае $d_{ij} = \infty$.

Имеется и другой, подобный путь вычисления транзитивного замыкания графа G в течение времени $\Theta(n^3)$, на практике позволяющий сэкономить время и память. Этот метод включает в себя подстановку логических операций $\vee$ (логическое ИЛИ) и $\wedge$ (логическое И) вместо используемых в алгоритме Флойда–Уоршелла арифметических операций min и $+$. Определим значение $t_{ij}^{(k)}$ для $i, j, k = 1, 2, \ldots, n$ равным 1, если в графе G существует путь из вершины i в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$; в противном случае эта величина равна 0. Конструируя транзитивное замыкание $G^* = (V, E^*)$, будем помещать ребро (i, j) в множество E^*

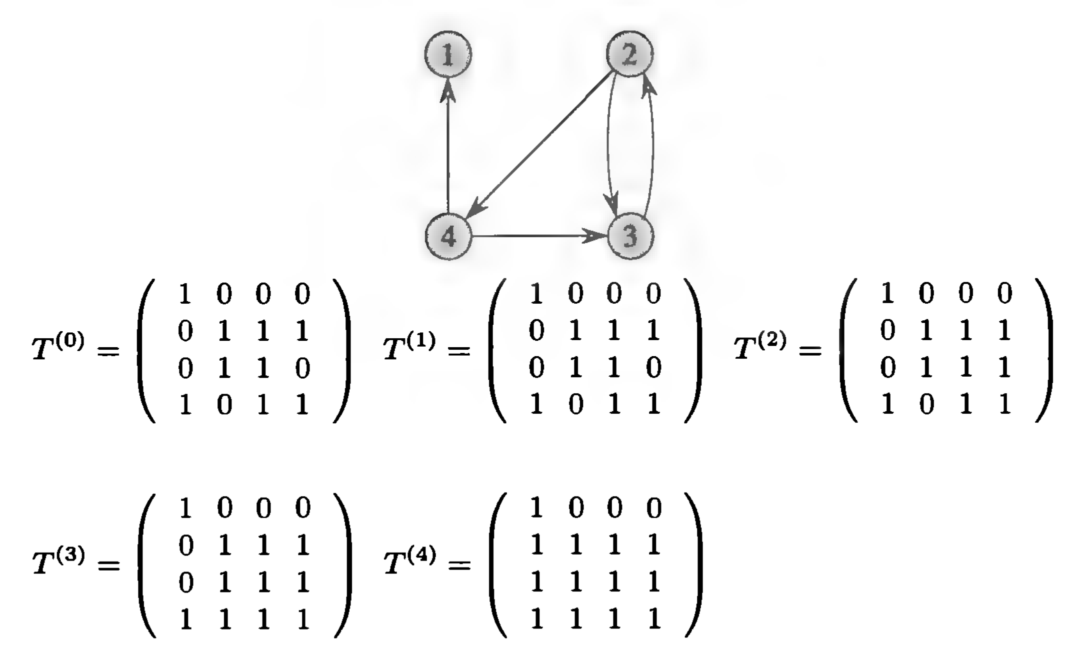
тогда и только тогда, когда $t_{ij}^{(k)} = 1$. Рекурсивное определение величины $t_{ij}^{(k)}$, построенное по аналогии с рекуррентным соотношением (25.5), имеет вид

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) . \qquad (25.8)$$

Как и в алгоритме Флойда–Уоршелла, мы вычисляем матрицы $T^{(k)} = \left(t_{ij}^{(k)}\right)$ в порядке возрастания k.

```
TRANSITIVE-CLOSURE(G)
n = |G.V|
Пусть T^(0) = (t_ij^(0)) — новая матрица размером n × n
for i = 1 to n
    for j = 1 to n
        if i == j или (i, j) ∈ G.E
            t_ij^(0) = 1
        else t_ij^(0) = 0
for k = 1 to n
    Пусть T^(k) = (t_ij^(k)) — новая матрица размером n × n
    for i = 1 to n
        for j = 1 to n
            t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

На рис. 25.5 показаны матрицы $T^{(k)}$, вычисленные процедурой TRANSITIVE-CLOSURE для приведенного графа. Время работы процедуры TRANSITIVE-CLO-



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Рис. 25.5. Ориентированный граф и матрицы $T^{(k)}$, вычисленные алгоритмом транзитивного замыкания.

SURE, как и время работы алгоритма Флойда–Уоршелла, равно $\Theta(n^3)$. Однако на некоторых компьютерах логические операции с однобитовыми величинами выполняются быстрее, чем арифметические операции со словами, представляющими целочисленные данные. Кроме того, поскольку в прямом алгоритме транзитивного замыкания используются только булевы, а не целые величины, ему требуется меньший объем памяти, чем алгоритму Флойда–Уоршелла. Объем сэкономленной памяти зависит от размера слова компьютера.

Упражнения

25.2.1

Примените алгоритм Флойда–Уоршелла ко взвешенному ориентированному графу, изображенному на рис. 25.2. Приведите матрицы $D^{(k)}$, полученные на каждой итерации внешнего цикла.

25.2.2

Покажите, как найти транзитивное замыкание с использованием методики из раздела 25.1.

25.2.3

Модифицируйте процедуру FLOYD-WARSHALL таким образом, чтобы в ней вычислялись матрицы $\Pi^{(k)}$ в соответствии с уравнениями (25.6) и (25.7). Дайте строгое доказательство того, что для всех $i \in V$ граф предшествования $G_{\pi,i}$ представляет собой дерево кратчайших путей с корнем i. (*Указание:* чтобы показать, что граф $G_{\pi,i}$ ациклический, сначала покажите, что из равенства $\pi_{ij}^{(k)} = l$ в соответствии с определением $\pi_{ij}^{(k)}$ следует, что $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$. Затем адаптируйте доказательство леммы 24.16.)

25.2.4

Как было показано, для работы алгоритма Флойда–Уоршелла требуется объем памяти, равный $\Theta(n^3)$, поскольку мы вычисляем величины $d_{uj}^{(k)}$ для $i, j, k = 1, 2, \ldots, n$. Покажите, что приведенная ниже процедура, в которой все верхние индексы просто опущены, корректна и что для ее работы требуется объем памяти $\Theta(n^2)$.

```
FLOYD-WARSHALL'(W)
n = W.rows
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d_ij = min(d_ij, d_ik + d_kj)
return D
```

24 Зак. 3726

25.2.5

Предположим, что мы модифицируем способ обработки равенства в уравнении (25.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Корректно ли такое альтернативное определение матрицы предшествования Π?

25.2.6

Как с помощью выходных данных алгоритма Флойда–Уоршелла установить наличие цикла с отрицательным весом?

25.2.7

В одном из способов, позволяющих восстановить в алгоритме Флойда–Уоршелла кратчайшие пути, используются величины $\phi_{ij}^{(k)}$ для $i, j, k = 1, 2, \ldots, n$, где $\phi_{ij}^{(k)}$ представляет собой промежуточную вершину с наибольшим номером, принадлежащую кратчайшему пути из вершины i в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$. Дайте рекурсивное определение величин $\phi_{ij}^{(k)}$, модифицируйте процедуру FLOYD-WARSHALL таким образом, чтобы в ней вычислялись эти величины, и перепишите процедуру PRINT-ALL-PAIRS-SHORTEST-PATH так, чтобы в качестве ее входных данных выступала матрица $\Phi = \left(\phi_{ij}^{(n)}\right)$. Чем матрица Φ похожа на таблицу s, которая используется в задаче о перемножении цепочки матриц из раздела 15.2?

25.2.8

Разработайте алгоритм, позволяющий вычислить транзитивное замыкание ориентированного графа $G = (V, E)$ за время $O(VE)$.

25.2.9

Предположим, что транзитивное замыкание ориентированного ациклического графа можно вычислить за время $f(|V|, |E|)$, где f — монотонно возрастающая функция от $|V|$ и $|E|$. Покажите, что в таком случае время поиска транзитивного замыкания $G^* = (V, E^*)$ ориентированного графа общего вида $G = (V, E)$ равно $f(|V|, |E|) + O(V + E^*)$.

25.3. Алгоритм Джонсона для разреженных графов

Алгоритм Джонсона позволяет найти кратчайшие пути между всеми парами вершин за время $O(V^2 \lg V + VE)$. Для разреженных графов в асимптотическом пределе он ведет себя лучше, чем алгоритм многократного возведения матриц в квадрат и алгоритм Флойда–Уоршелла. Этот алгоритм либо возвращает матрицу, содержащую веса кратчайших путей для всех пар вершин, либо выводит сообщение о том, что входной граф содержит цикл с отрицательным весом. В ал-

горитме Джонсона используются подпрограммы, в которых реализованы алгоритмы Дейкстры и Беллмана–Форда, описанные в главе 24.

В алгоритме Джонсона используется метод ***изменения веса*** (reweighting), работающий следующим образом. Если веса всех ребер w в графе $G = (V, E)$ неотрицательны, можно найти кратчайшие пути между всеми парами вершин, по разу запустив алгоритм Дейкстры для каждой вершины. Если неубывающая очередь с приоритетами реализована в виде пирамиды Фибоначчи, то время работы такого алгоритма будет равно $O(V^2 \lg V + VE)$. Если же в графе G содержатся ребра с отрицательным весом, но отсутствуют циклы с отрицательным весом, можно вычислить новое множество неотрицательных весов ребер, позволяющее воспользоваться тем же методом. Новое множество весов ребер $\widehat{w}$ должно удовлетворять двум важным свойствам.

1. Для всех пар вершин $u, v \in V$ путь p является кратчайшим путем из вершины u в вершину v с использованием весовой функции w тогда и только тогда, когда p является также кратчайшим путем из вершины u в вершину v с весовой функцией $\widehat{w}$.
2. Для всех ребер (u, v) новый вес $\widehat{w}(u, v)$ неотрицателен.

Как мы вскоре увидим, предварительную обработку графа G с целью определить новую весовую функцию $\widehat{w}$ можно выполнить за время $O(VE)$.

Сохранение кратчайших путей

Приведенная ниже лемма показывает, как легко организовать изменение весов, удовлетворяющее первому из сформулированных выше свойств. Значения весов кратчайших путей, полученные с помощью весовой функции w, обозначены как δ, а веса кратчайших путей, полученных с помощью весовой функции $\widehat{w}$, — как $\widehat{\delta}$.

Лемма 25.1 (Изменение весов сохраняет кратчайшие пути)

Пусть для заданного взвешенного ориентированного графа $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ функция $h : V \to \mathbb{R}$ — произвольная функция, отображающая вершины на действительные числа. Для каждого ребра $(u, v) \in E$ определим

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) . \tag{25.9}$$

Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$ — произвольный путь из вершины v_0 в вершину v_k. Путь p является кратчайшим путем с весовой функцией w тогда и только тогда, когда он является кратчайшим путем с весовой функцией $\widehat{w}$, т.е. равенство $w(p) = \delta(v_0, v_k)$ равносильно равенству $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Кроме того, граф G содержит цикл с отрицательным весом при использовании весовой функции w тогда и только тогда, когда он содержит цикл с отрицательным весом при использовании весовой функции $\widehat{w}$.

Доказательство. Сначала покажем, что

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k) . \tag{25.10}$$

Мы имеем

$$\begin{aligned}\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(в силу телескопичности суммы)} \\ &= w(p) + h(v_0) - h(v_k)\ .\end{aligned}$$

Следовательно, для любого пути p из вершины v_0 в вершину v_k справедливо соотношение $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Поскольку $h(v_0)$ и $h(v_k)$ не зависят от пути, то если один путь из v_0 в v_k короче другого при использовании весовой функции w, то он будет короче и при использовании весовой функции $\widehat{w}$. Таким образом, $w(p) = \delta(v_0, v_k)$ тогда и только тогда, когда $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

Наконец покажем, что граф G имеет цикл с отрицательным весом с использованием весовой функции w тогда и только тогда, когда он имеет такой цикл с использованием весовой функции $\widehat{w}$. Рассмотрим произвольный цикл $c = \langle v_0, v_1, \ldots, v_k\rangle$, где $v_0 = v_k$. В соответствии с уравнением (25.10) выполняется соотношение

$$\begin{aligned}\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c)\ ,\end{aligned}$$

а следовательно, вес цикла c будет отрицательным с использованием весовой функции w тогда и только тогда, когда он отрицательный с использованием весовой функции $\widehat{w}$. ■

Генерация неотрицательных весов путем их изменения

Следующая наша цель — обеспечить выполнение второго свойства: нужно, чтобы величина $\widehat{w}(u, v)$ была неотрицательной для всех ребер $(u, v) \in E$. Для данного взвешенного ориентированного графа $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$ мы создадим новый граф $G' = (V', E')$, где $V' = V \cup \{s\}$ для некоторой новой вершины $s \notin V$ и $E' = E \cup \{(s, v) : v \in V\}$. Расширим весовую функцию w таким образом, чтобы для всех вершин $v \in V$ выполнялось равенство $w(s, v) = 0$. Заметим, что поскольку в вершину s не входит ни одно ребро, эту вершину не содержит ни один кратчайший путь графа G', отличный от того, который исходит из s. Кроме того, граф G' не содержит циклов с отрицательным весом тогда и только тогда, когда таких циклов не содержит граф G. На рис. 25.6, (а) показан граф G', соответствующий графу G на рис. 25.1.

Теперь предположим, что графы G и G' не содержат циклов с отрицательным весом. Определим для всех вершин $v \in V'$ величину $h(v) = \delta(s, v)$. Согласно

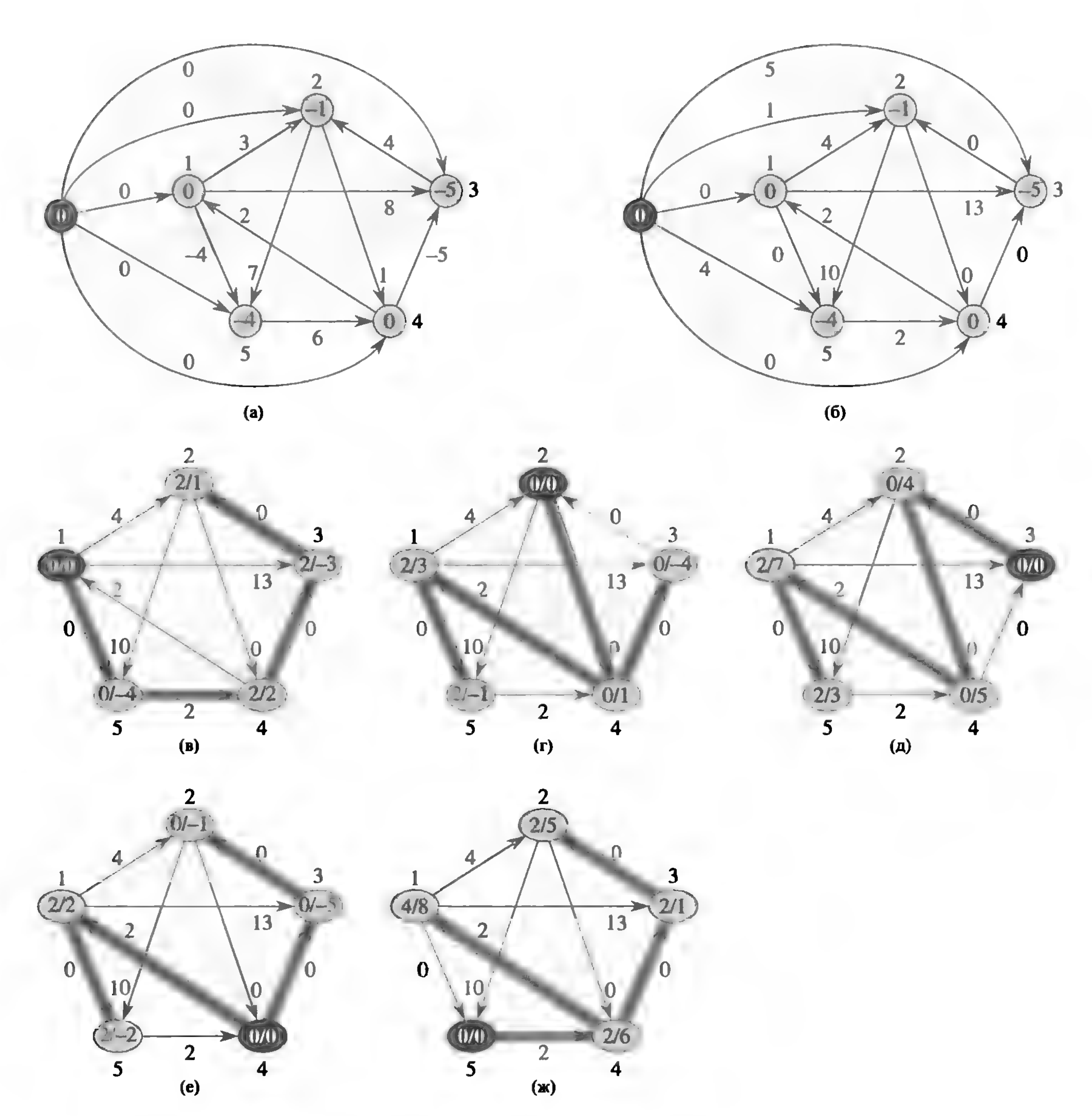


Рис. 25.6. Работа алгоритма Джонсона, предназначенного для поиска кратчайших путей между всеми парами вершин, с графом, изображенным на рис. 25.1. **(а)** Граф G' с исходной весовой функцией w. Новая вершина s указана черным цветом. В каждой вершине v показано значение $h(v) = \delta(s, v)$. **(б)** После изменения веса каждого ребра (u, v) с использованием весовой функции $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(в)–(ж)** Результат применения алгоритма Дейкстры для каждой вершины G с использованием весовой функции $\widehat{w}$. В каждой части источник u указан черным цветом, а заштрихованные ребра образуют дерево кратчайших путей, вычисленное алгоритмом. В каждой вершине v показаны значения $\widehat{\delta}(u, v)$ и $\delta(u, v)$, разделенные косой чертой. Значение $d_{uv} = \delta(u, v)$ равно $\widehat{\delta}(u, v) + h(v) - h(u)$.

неравенству треугольника (лемма 24.10) для всех ребер $(u, v) \in E'$ выполняется соотношение $h(v) \leq h(u) + w(u, v)$. Таким образом, если мы определим новые веса $\widehat{w}$ в соответствии с уравнением (25.9), то получим $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, так что второе свойство удовлетворяется. На рис. 25.6, (б) показан граф G', полученный в результате переопределения весов ребер графа, изображенного на рис. 25.6, (а).

Вычисление кратчайших путей между всеми парами вершин

В алгоритме Джонсона, предназначенном для вычисления кратчайших путей между всеми парами вершин, в качестве подпрограмм используются алгоритм Беллмана–Форда (раздел 24.1) и алгоритм Дейкстры (раздел 24.3). Предполагается, что ребра хранятся в виде списков смежных вершин. Этот алгоритм возвращает обычную матрицу $D = d_{ij}$ размером $|V| \times |V|$, где $d_{ij} = \delta(i, j)$, или выдает сообщение о том, что входной граф содержит цикл с отрицательным весом. Предполагается, что вершины пронумерованы от 1 до $|V|$, что типично для алгоритмов, предназначенных для поиска кратчайших путей между всеми парами вершин.

JOHNSON(G, w)

1 Вычислить G', где $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, и
    $w(s, v) = 0$ для всех $v \in G.V$
2 **if** BELLMAN-FORD(G', w, s) == FALSE
3   print "входной граф содержит цикл с отрицательным весом"
4 **else for** каждой вершины $v \in G'.V$
5   установить $h(v)$ равным значению $\delta(s, v)$,
            вычисленному алгоритмом Беллмана–Форда
6   **for** каждого ребра $(u, v) \in G'.E$
7         $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$
8   Пусть $D = (d_{uv})$ — новая матрица размером $n \times n$
9   **for** каждой вершины $u \in G.V$
10   выполнить DIJKSTRA$(G, \widehat{w}, u)$ для вычисления
            $\widehat{\delta}(u, v)$ для всех $v \in G.V$
11   **for** каждой вершины $v \in G.V$
12             $d_{uv} = \widehat{\delta}(u, v) + h(v) - h(u)$
13   **return** D

В этом коде просто выполняются описанные ранее действия. В строке 1 строится граф G'. В строке 2 выполняется алгоритм Беллмана–Форда с входным графом G', весовой функцией w и истоком s. Если граф G', а следовательно и граф G, содержат цикл с отрицательным весом, об этом выводится сообщение в строке 3. В строках 4–12 предполагается, что граф G' не содержит циклов с отрицательным весом. В строках 4 и 5 для всех вершин $v \in V'$ величине $h(v)$ присваивается вес кратчайшего пути $\delta(s, v)$, вычисленный алгоритмом Беллмана–Форда. В строках 6 и 7 для каждого ребра вычисляется новый вес $\widehat{w}$. Для каждой пары

вершин $u, v \in V$ цикл **for** в строках 9–12 вычисляет вес кратчайшего пути $\widehat{w}(u, v)$ с помощью однократного вызова алгоритма Дейкстры для каждой вершины из множества V. В строке 12 в элемент матрицы d_{uv} заносится корректный вес кратчайшего пути $\delta(u, v)$, вычисленный с помощью уравнения (25.10). Наконец в строке 13 возвращается вычисленная матрица D. Работа алгоритма Джонсона проиллюстрирована на рис. 25.6.

Если неубывающая очередь с приоритетами реализована в алгоритме Дейкстры в виде пирамиды Фибоначчи, то время работы алгоритма Джонсона равно $O(V^2 \lg V + VE)$. Более простая реализация неубывающей очереди с приоритетами приводит к тому, что время работы становится равным $O(VE \lg V)$, но для разреженных графов эта величина в асимптотическом пределе ведет себя лучше, чем время работы алгоритма Флойда–Уоршелла.

Упражнения

25.3.1

Вычислите с помощью алгоритма Джонсона кратчайшие пути между всеми парами вершин в графе, изображенном на рис. 25.2. Приведите значения h и $\widehat{w}$, вычисленные этим алгоритмом.

25.3.2

Зачем в множество V добавляется новая вершина s, в результате чего создается множество V'?

25.3.3

Предположим, что для всех ребер $(u, v) \in E$ выполняется неравенство $w(u, v) \geq 0$. Как между собой взаимосвязаны весовые функции w и $\widehat{w}$?

25.3.4

Профессор утверждает, что изменить веса ребер можно проще, чем это делается в алгоритме Джонсона. Полагая $w^* = \min_{(u,v) \in E} \{w(u, v)\}$, просто определим $\widehat{w}(u, v) = w(u, v) - w^*$ для всех ребер $(u, v) \in E$. Где кроется ошибка в методе, предложенном профессором?

25.3.5

Предположим, что алгоритм Джонсона выполняется для ориентированного графа G с весовой функцией w. Покажите, что если граф G содержит цикл c с нулевым весом, то для всех ребер (u, v) этого цикла $\widehat{w}(u, v) = 0$.

25.3.6

Профессор утверждает, что нет необходимости создавать новую вершину-исток в строке 1 алгоритма JOHNSON. Профессор считает, что вместо этого можно использовать $G' = G$, а в роли вершины s использовать произвольную вершину. Приведите пример взвешенного ориентированного графа G, для которого воплощение идеи профессора в алгоритме JOHNSON даст неправильный ответ. Затем покажите, что если граф G сильно связный (каждая его вершина достижима из

любой другой вершины), то результаты работы алгоритма JOHNSON, модифицированного профессором, будут верны.

Задачи

25.1. Транзитивное замыкание динамического графа

Предположим, что нужно поддерживать транзитивное замыкание ориентированного графа $G = (V, E)$ по мере добавления ребер в множество E. Другими словами, после добавления каждого ребра нужно обновить транзитивное замыкание добавленных до этого времени ребер. Предположим, что граф G изначально не содержит ребер и что транзитивное замыкание должно быть представлено в виде булевой матрицы.

а. Покажите, каким образом транзитивное замыкание $G^* = (V, E^*)$ графа $G = (V, E)$ можно обновить в течение времени $O(V^2)$ после добавления нового ребра в граф G.

б. Приведите пример графа G и ребра e, такой, что обновление транзитивного замыкания после добавления ребра e в граф G будет выполняться в течение времени $\Omega(V^2)$ независимо от используемого алгоритма.

в. Разработайте эффективный алгоритм обновления транзитивного замыкания по мере добавления ребер в граф. Для любой последовательности n добавлений общее время работы этого алгоритма должно быть равно $\sum_{i=1}^{n} t_i = O(V^3)$, где t_i — время, необходимое для обновления транзитивного замыкания при добавлении i-го ребра. Докажите, что в вашем алгоритме достигается указанная граница времени работы.

25.2. Кратчайшие пути в ϵ-плотном графе

Граф $G = (V, E)$ называется ***ϵ-плотным*** (ϵ-dense), если $|E| = \Theta(V^{1+\epsilon})$ для некоторой константы ϵ в диапазоне $0 < \epsilon \leq 1$. Если в алгоритмах, предназначенных для поиска кратчайших путей в ϵ-плотных графах, воспользоваться d-арными неубывающими пирамидами (см. задачу 6.2), то время их выполнения может быть сопоставимо со временем работы алгоритмов, основанных на применении пирамиды Фибоначчи. При этом удается обойтись без сложных структур данных.

а. Чему равно асимптотическое время работы процедур INSERT, EXTRACT-MIN и DECREASE-KEY как функции от кратности d и количества n элементов d-арной неубывающей пирамиды? Чему равно время работы этих алгоритмов, если выбрать $d = \Theta(n^\alpha)$, где $0 < \alpha \leq 1$ — некоторая константа? Сравните эти времена работы с амортизированными стоимостями этих операций для пирамиды Фибоначчи.

б. Покажите, как за время $O(E)$ вычислить кратчайшие пути из единого истока в ϵ-плотном ориентированном графе $G = (V, E)$, в котором отсутствуют ребра с отрицательным весом. (*Указание:* выберите величину d как функцию от ϵ.)

в. Покажите, как за время $O(VE)$ решить задачу поиска кратчайших путей между всеми парами вершин в ϵ-плотном ориентированном графе $G = (V, E)$, в котором отсутствуют ребра с отрицательным весом.

г. Покажите, как за время $O(VE)$ решить задачу поиска кратчайших путей между всеми парами вершин в ϵ-плотном ориентированном графе $G = (V, E)$, в котором допускается наличие ребер с отрицательным весом, но отсутствуют циклы с отрицательным весом.

Заключительные замечания

Неплохое обсуждение задачи о поиске кратчайших путей между всеми парами вершин содержится в книге Лоулера (Lawler) [223], хотя в ней и не анализируются решения для разреженных графов. Алгоритм перемножения матриц он считает результатом народного творчества. Алгоритм Флойда–Уоршелла был предложен Флойдом (Floyd) [104], который основывался на теореме Уоршелла (Warshall) [347], в которой описывается, как найти транзитивное замыкание булевых матриц. Алгоритм Джонсона (Johnson) взят из статьи [191].

Ряд исследователей предложили улучшенные алгоритмы, предназначенные для поиска кратчайших путей с помощью умножения матриц. Фридман (Fredman) [110] показал, что задачу о поиске кратчайшего пути между всеми парами вершин можно решить, выполнив $O(V^{5/2})$ операций сравнения суммарных весов ребер; в результате получится алгоритм, время работы которого равно $O(V^3(\lg\lg V/\lg V)^{1/3})$, что несколько лучше соответствующей величины для алгоритма Флойда–Уоршелла. Хан (Han) [158] сумел сократить время работы до $O(V^3(\lg\lg V/\lg V)^{5/4})$. Еще одно направление исследований показывает, что к задаче о поиске кратчайших путей между всеми парами вершин можно применить алгоритмы быстрого умножения матриц (см. заключительные замечания к главе 4). Пусть $O(n^\omega)$ — время работы самого производительного алгоритма, предназначенного для перемножения матриц размером $n \times n$; в настоящее время $\omega < 2.376$ [77]. Галил (Galil) и Маргалит (Margalit) [122, 123], а также Зайдель (Seidel) [306] разработали алгоритмы, решающие задачу о поиске кратчайших путей между всеми парами вершин в неориентированных невзвешенных графах за время $(V^\omega p(V))$, где $p(n)$ обозначает функцию, полилогарифмически ограниченную по n. В плотных графах время работы этих алгоритмов меньше величины $O(VE)$, необходимой для $|V|$ поисков в ширину. Некоторые исследователи расширили эти результаты и разработали алгоритмы, предназначенные для поиска кратчайших путей между всеми парами вершин в неориентированных графах с целочисленными весами ребер в диапазоне $\{1, 2, \ldots, W\}$. Среди таких алго-

ритмов быстрее всего в асимптотическом пределе ведет себя алгоритм Шошана (Shoshan) и Цвика (Zwick) [314], время работы которого равно $O(WV^{\omega}p(VW))$.

Каргер (Karger), Коллер (Koller) и Филлипс (Phillips) [195], а также независимо Мак-Геч (McGeoch) [245] дали временну́ю границу, зависящую от E^*, подмножества ребер из множества E, входящих в некоторый кратчайший путь. Для заданного графа с неотрицательными весами ребер эти алгоритмы выполняются за время $O(VE^* + V^2 \lg V)$. Это лучший показатель, чем $|V|$-кратное выполнение алгоритма Дейкстры, если $|E^*| = o(E)$.

Басвана (Baswana), Харихаран (Hariharan) и Сен (Sen) [32] исследовали декрементные алгоритмы для поддержки кратчайших путей между всеми парами вершин и информации о транзитивном замыкании. Декрементные алгоритмы позволяют выполнять последовательность чередующихся удалений ребер и запросов; задаче же 25.1, в которой ребра вставляются, напротив, требуется инкрементный алгоритм. Эти алгоритмы Басваны, Харихана и Сена рандомизированные, и, когда путь существует, их алгоритм для транзитивного замыкания может ошибаться с вероятностью $1/n^c$ для произвольного $c > 0$. Время выполнения запросов с высокой вероятностью равно $O(1)$. Что касается транзитивного замыкания, то амортизированное время для каждого обновления составляет $O(V^{4/3} \lg^{1/3} V)$. Для кратчайших путей между всеми парами вершин время обновления зависит от запросов. Для запросов, которые должны выдавать только веса кратчайших путей, амортизированное время одного обновления составляет $O(V^3/E \lg^2 V)$. При поиске фактического кратчайшего пути амортизированное время обновления равно $\min(O(V^{3/2}\sqrt{\lg V}), O(V^3/E \lg^2 V))$. Деметреску (Demetrescu) и Итальяно (Italiano) [83] показали, как работать с операциями обновления и запросами, когда ребра и вставляются, и удаляются, и при этом каждое ребро имеет ограниченный диапазон возможных действительных значений.

Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] дали определение алгебраической структуры, известной как "замкнутое полукольцо". Эта структура служит общим каркасом для решения задач о поиске путей в ориентированных графах. Алгоритм Флойда–Уоршелла и описанный в разделе 25.2 алгоритм поиска транзитивного замыкания – примеры алгоритмов поиска путей между всеми парами вершин, основанных на замкнутых полукольцах. Маггс (Maggs) и Плоткин (Plotkin) [239] показали, как искать минимальные остовные деревья с помощью замкнутых полуколец.

Глава 26. Задача о максимальном потоке

Так же, как дорожную карту можно смоделировать ориентированным графом, чтобы найти кратчайший путь из одной точки в другую, так и ориентированный граф можно интерпретировать как некоторую транспортную сеть и использовать для решения задач о потоках вещества в системе трубопроводов. Представим, что некоторый продукт передается по системе от истока, где данный продукт производится, к стоку, где он потребляется. Исток производит продукт с некоторой постоянной скоростью, а сток с той же скоростью его потребляет. Интуитивно потоком продукта в любой точке системы является скорость его движения. С помощью транспортных сетей можно моделировать течение жидкостей по трубопроводам, движение деталей на сборочных линиях, передачу тока по электрическим сетям, информации — по информационным сетям и т.д.

Каждое ориентированное ребро сети можно рассматривать как канал, по которому движется продукт. Каждый канал имеет заданную пропускную способность, которая характеризует максимальную скорость перемещения продукта по каналу, например 200 литров жидкости в минуту для трубопровода или 20 ампер — для провода электрической цепи. Вершины являются точками пересечения каналов; через вершины, отличные от истока и стока, продукт проходит, не накапливаясь. Иными словами, скорость поступления продукта в вершину должна быть равна скорости его удаления из вершины. Это свойство называется свойством сохранения потока; в случае передачи тока по электрическим цепям ему соответствует закон Кирхгофа.

В задаче о максимальном потоке мы хотим найти максимальную скорость пересылки продукта от истока к стоку, при которой не будут нарушаться ограничения пропускной способности. Это одна из простейших задач, возникающих в транспортных сетях, и, как будет показано в данной главе, существуют эффективные алгоритмы ее решения. Более того, основные методы, используемые в алгоритмах решения задач о максимальном потоке, можно применять для решения других задач, связанных с транспортными сетями.

В данной главе предлагается два общих метода решения задачи о максимальном потоке. В разделе 26.1 формализуются понятия транспортных сетей и потоков, а также дается формальное определение задачи о максимальном потоке. В разделе 26.2 описывается классический метод Форда–Фалкерсона (Ford–Fulkerson) для поиска максимального потока. В качестве приложения данного метода в разделе 26.3 осуществляется поиск максимального паросочетания в неори-

ентированном двудольном графе. В разделе 26.4 описывается метод "проталкивания предпотока" ("push-relabel"), который лежит в основе многих наиболее быстрых алгоритмов для решения задач в транспортных сетях. В разделе 26.5 рассматривается еще один алгоритм, время работы которого составляет $O(V^3)$. Хотя он и не является самым быстрым известным алгоритмом, зато позволяет проиллюстрировать некоторые методы, используемые в асимптотически более быстрых алгоритмах, и на практике является достаточно эффективным.

26.1. Транспортные сети

В данном разделе мы дадим определение транспортных сетей в терминах теории графов, обсудим их свойства, дадим точное определение задачи о максимальном потоке, а также введем некие полезные обозначения.

Транспортные сети и потоки

Транспортная сеть (flow network) $G = (V, E)$ представляет собой ориентированный граф, в котором каждое ребро $(u, v) \in E$ имеет неотрицательную ***пропускную способность*** (capacity) $c(u, v) \geq 0$. Далее мы потребуем, чтобы в случае, если E содержит ребро (u, v), обратного ребра (v, u) не было (вскоре мы увидим, как обойти это ограничение). Если $(u, v) \notin E$, то для удобства определим $c(u, v) = 0$, а также запретим петли. В транспортной сети выделяются две вершины: ***исток*** (source) s и ***сток*** (sink) t. Для удобства предполагается, что каждая вершина лежит на неком пути от истока к стоку, т.е. для любой вершины $v \in V$ транспортная сеть содержит путь $s \leadsto v \leadsto t$. Таким образом, граф является связным и, поскольку каждая вершина, отличная от s, содержит как минимум одно входящее ребро, $|E| \geq |V| - 1$. На рис. 26.1 показан пример транспортной сети.

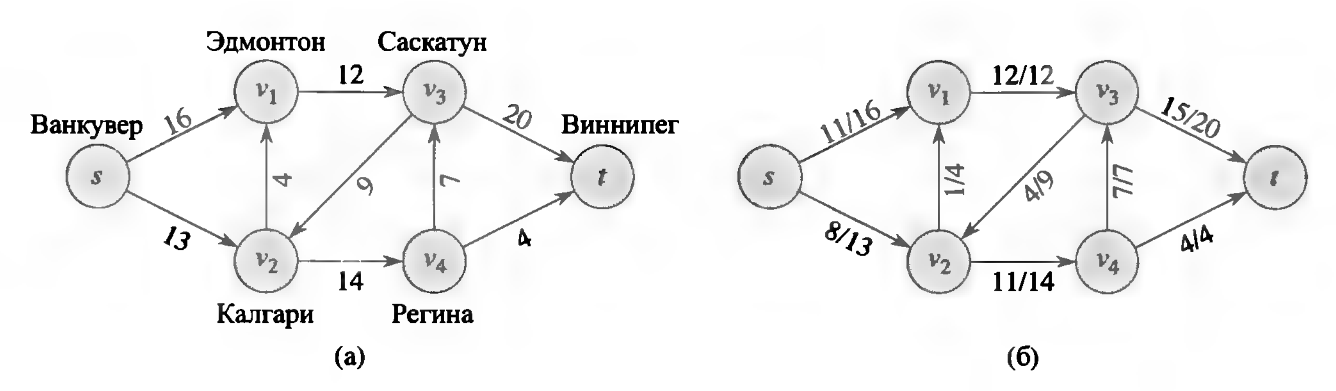


Рис. 26.1. **(а)** Транспортная сеть $G = (V, E)$ для задачи о грузовых перевозках компании Lucky Puck. Истоком s является фабрика в Ванкувере, а стоком t — склад в Виннипеге. Шайбы доставляются через промежуточные города, но за день из города u в город v можно отправить только $c(u, v)$ ящиков. На рисунке указана пропускная способность каждого ребра сети. **(б)** Поток f в транспортной сети G со значением $|f| = 19$. Каждое ребро (u, v) имеет метку $f(u, v)/c(u, v)$ (косая черта используется только для того, чтобы отделить поток от пропускной способности, и не обозначает деление).

Теперь мы готовы дать более формальное определение потоков. Пусть $G = (V, E)$ — транспортная сеть с функцией пропускной способности c. Пусть s является истоком, а t — стоком. ***Потоком*** (flow) в G является действительная функция $f : V \times V \to \mathbb{R}$, удовлетворяющая следующим двум условиям.

Ограничение пропускной способности. Для всех $u, v \in V$ должно выполняться $0 \leq f(u,v) \leq c(u,v)$.

Сохранение потока. Для всех $u \in V - \{s,t\}$ должно выполняться

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v) .$$

Когда $(u,v) \notin E$, потока из u в v быть не может, так что $f(u,v) = 0$.

Неотрицательную величину $f(u,v)$ мы называем потоком из вершины u в вершину v. ***Величина*** (value) $|f|$ потока f определяется как

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) , \tag{26.1}$$

т.е. как суммарный поток, выходящий из истока, минус входящий в него. (Здесь запись $|\cdot|$ означает величину потока, а не абсолютное значение или мощность.) Обычно транспортная сеть не имеет ребер, входящих в исток, и поток в исток, задаваемый суммой $\sum_{v \in V} f(v,s)$, равен 0. Однако мы включаем его, поскольку позже в этой главе при рассмотрении остаточных сетей потоки в исток станут важными. В ***задаче о максимальном потоке*** (maximum flow problem) дана некоторая транспортная сеть G с истоком s и стоком t, и необходимо найти поток максимальной величины.

Перед тем как рассматривать пример задачи о потоке в сети, кратко проанализируем определение потока и два его свойства. Ограничение пропускной способности просто гласит, что поток из одной вершины в другую должен быть неотрицательным и не должен превышать заданную пропускную способность ребра. Свойство сохранения потока утверждает, что суммарный поток, входящий в вершину, не являющуюся истоком или стоком, должен быть равен суммарному выходящему потоку, т.е., иначе говоря, что в вершину "втекает", то из нее тут же и "вытекает".

Пример потока

С помощью транспортной сети можно моделировать задачу о грузовых перевозках, представленную на рис. 26.1, (а). У компании Lucky Puck в Ванкувере есть фабрика (исток s), производящая хоккейные шайбы, а в Виннипеге — есть склад (сток t), где эти шайбы хранятся. Компания арендует места на грузовиках других фирм для доставки шайб с фабрики на склад. Поскольку грузовики ездят по определенным маршрутам (ребрам) между городами (вершинами) и имеют ограниченную грузоподъемность, компания Lucky Puck может перевозить не более $c(u,v)$ ящиков в день между каждой парой городов u и v, как показано

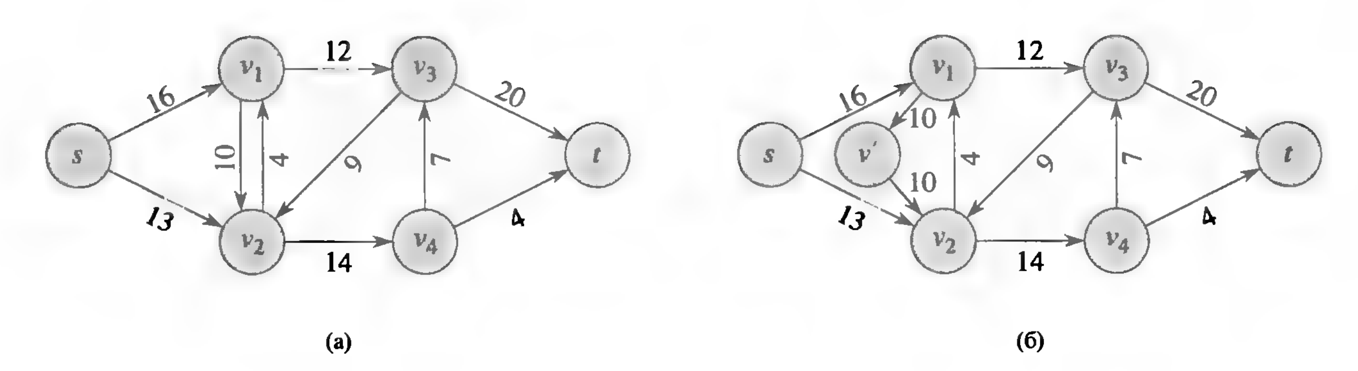


Рис. 26.2. Преобразование сети с антипараллельными ребрами в эквивалентную сеть без таковых. **(а)** Транспортная сеть, содержащая как ребро (v_1, v_2), так и ребро (v_2, v_1). **(б)** Эквивалентная сеть без антипараллельных ребер. Добавлена новая вершина v', а ребро (v_1, v_2) заменено парой ребер (v_1, v') и (v', v_2) с одной и той же пропускной способностью (v_1, v_2).

на рис. 26.1, (а). Компания Lucky Puck не может повлиять на маршруты и пропускную способность, т.е. не может менять транспортную сеть, представленную на рис. 26.1, (а). Ее задача — определить, какое наибольшее количество p ящиков в день можно отгружать, и затем производить именно такое количество, поскольку не имеет смысла производить шайб больше, чем можно отправить на склад. Для компании не важно, сколько времени займет доставка конкретной шайбы с фабрики на склад, она заботится только о том, чтобы p ящиков в день отправлялось с фабрики и p ящиков в день прибывало на склад.

Можно смоделировать потоком в данной сети "поток" отгрузок, поскольку число ящиков, отгружаемых ежедневно из одного города в другой, подчиняется ограничению пропускной способности. Кроме того, должно соблюдаться условие сохранения потока, поскольку в стационарном состоянии скорость ввоза шайб в некоторый промежуточный город должна быть равна скорости их вывоза. В противном случае ящики станут накапливаться в промежуточных городах.

Моделирование задач с антипараллельными ребрами

Предположим, что фирма-перевозчик предлагает Lucky Puck перевозку десяти ящиков в грузовике, идущем из Эдмонтона в Калгари. Представляется естественным добавить эту возможность в наш пример и получить транспортную сеть, показанную на рис. 26.2, (а). Однако здесь возникает одна проблема: эта сеть нарушает исходное предположение о том, что если $(v_1, v_2) \in E$, то $(v_2, v_1) \notin E$. Два ребра, (v_1, v_2) и (v_2, v_1), называются ***антипараллельными*** (antiparallel). Таким образом, если мы хотим моделировать задачу о потоке при наличии антипараллельных ребер, сеть следует преобразовать в эквивалентную, не содержащую антипараллельных ребер. На рис. 26.2, (б) показана такая эквивалентная сеть. Мы выбираем одно из двух антипараллельных ребер, в данном случае ребро (v_1, v_2), и разбиваем его, добавляя новую вершину v' и заменяя ребро (v_1, v_2) парой ребер (v_1, v') и (v', v_2). Мы также устанавливаем пропускную способность обоих новых ребер равной пропускной способности исходного ребра. Полученная в результате сеть удовлетворяет свойству, согласно которому при наличии некоторого ребра в сети встречное ребро в ней должно отсутствовать. В упр. 26.1.1 требует-

ся доказать, что полученная в результате описанных действий сеть эквивалентна исходной.

Таким образом, как видим, реальные задачи о потоках могут быть более естественно смоделированы сетями с антипараллельными ребрами. Однако поскольку более удобным оказывается запрет на антипараллельные ребра, описанная процедура дает нам простой способ преобразовать сеть с антипараллельными ребрами в эквивалентную сеть без таковых.

Сети с несколькими истоками и стоками

В задаче о максимальном потоке может быть несколько истоков и стоков. Например, у компании Lucky Puck может быть m фабрик $\{s_1, s_2, \ldots, s_m\}$ и n складов $\{t_1, t_2, \ldots, t_n\}$, как показано на рис. 26.3, (а), на котором приведен пример транспортной сети с пятью истоками и тремя стоками. К счастью, эта задача не сложнее, чем обычная задача о максимальном потоке.

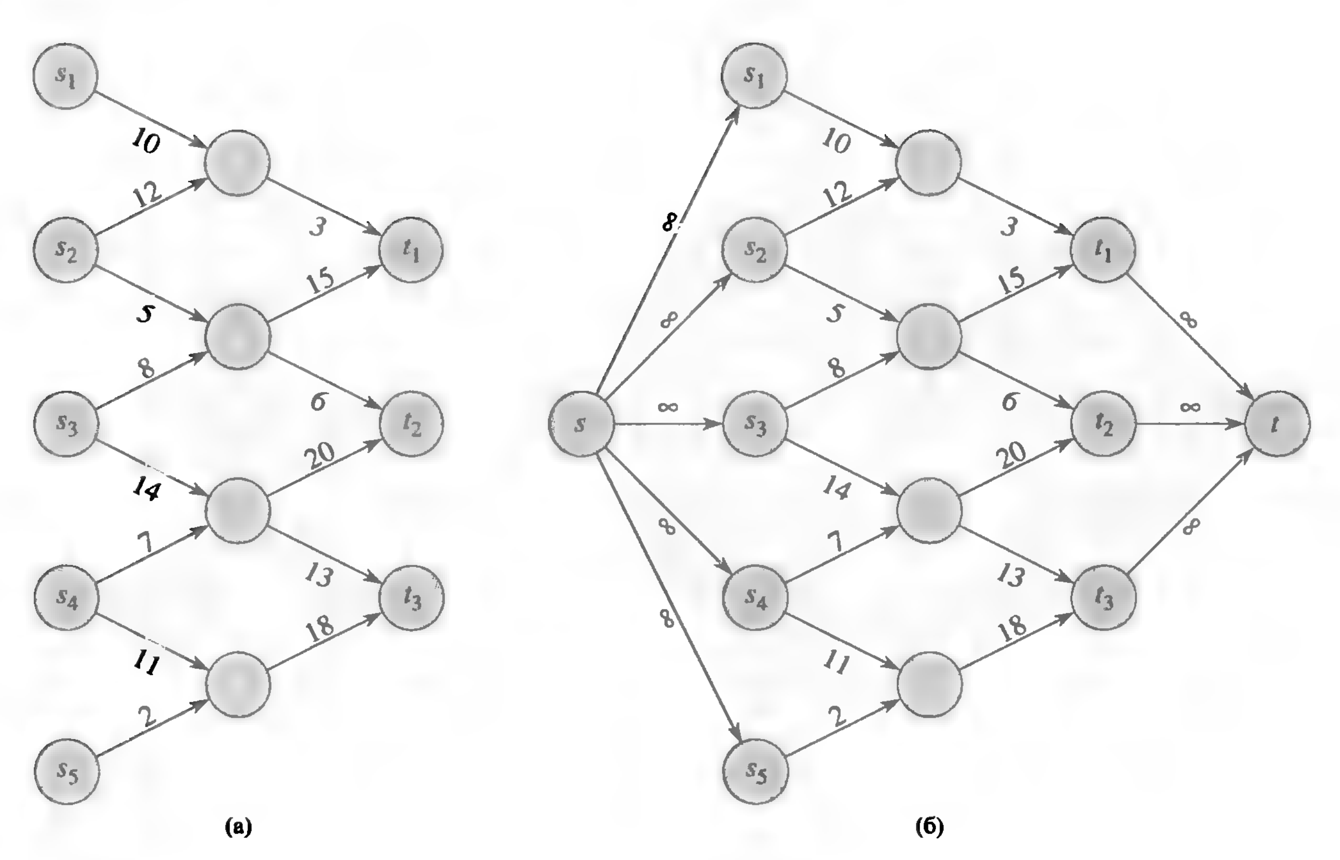


Рис. 26.3. Преобразование задачи о максимальном потоке с несколькими истоками и стоками в задачу с одним истоком и одним стоком. **(а)** Транспортная сеть с пятью истоками $S = \{s_1, s_2, s_3, s_4, s_5\}$ и тремя стоками $T = \{t_1, t_2, t_3\}$. **(б)** Эквивалентная сеть с одним истоком и одним стоком. Мы добавляем фиктивный исток s и ребра с бесконечной пропускной способностью от s до каждого из исходных истоков. Кроме того, мы добавляем фиктивный сток t и ребра с бесконечной пропускной способностью из каждого из исходных истоков в t.

Задача определения максимального потока в сети с несколькими истоками и несколькими стоками сводится к обычной задаче о максимальном потоке. На рис. 26.3, (б) показано, как сеть, представленную на рис. 26.3, (а), можно превратить в обычную транспортную сеть с одним истоком и одним стоком. Для этого в сеть добавляются ***фиктивный исток*** (supersource) s и ориентированные реб-

ра (s, s_i) с пропускной способностью $c(s, s_i) = \infty$ для каждого $i = 1, 2, \ldots, m$. Точно так же создается ***фиктивный сток*** (supresink) t и добавляются ориентированные ребра (t_i, t) с $c(t_i, t) = \infty$ для каждого $i = 1, 2, \ldots, n$. Интуитивно понятно, что любой поток в сети на рис. 26.3, (а) соответствует потоку в сети на рис. 26.3, (б), и наоборот. Единственный исток s просто обеспечивает поток любой требуемой величины к истокам s_i, а единственный сток t аналогичным образом потребляет поток любой желаемой величины от множественных стоков t_i. В упр. 26.1.2 предлагается формально доказать эквивалентность этих двух задач.

Упражнения

26.1.1

Покажите, что разделение ребра в транспортной сети дает эквивалентную сеть. Говоря более формально, предположим, что транспортная сеть G содержит ребро (u, v), и мы создаем транспортную сеть G' путем добавления новой вершины x и замены (u, v) новыми ребрами (u, x) и (x, v), такими, что $c(u, x) = c(x, v) = c(u, v)$. Покажите, что максимальный поток в G' имеет ту же величину, что и в G.

26.1.2

Распространите свойства потока и определения на задачу с несколькими истоками и стоками. Покажите, что любой поток в сети с несколькими истоками и стоками соответствует потоку той же величины в сети с единственным истоком и стоком, получаемой путем добавления фиктивных истока и стока, и наоборот.

26.1.3

Предположим, что транспортная сеть $G = (V, E)$ нарушает предположение о том, что в сети имеются пути $s \leadsto v \leadsto t$ для всех вершин $v \in V$. Пусть u представляет собой вершину, для которой такого пути $s \leadsto u \leadsto t$ нет. Покажите, что в G должен существовать максимальный поток f, такой, что $f(u, v) = f(v, u) = 0$ для всех вершин $v \in V$.

26.1.4

Пусть f является потоком в сети, и пусть α — действительное число. ***Скалярным произведением потока*** (scalar flow product), обозначаемым как αf, является функция, отображающая $V \times V$ на $\mathbb{R}$ и определенная следующим образом:

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Докажите, что потоки в сети образуют ***выпуклое множество***, т.е. покажите, что если f_1 и f_2 являются потоками, то потоком является и $\alpha f_1 + (1 - \alpha) f_2$ для всех α из диапазона $0 \leq \alpha \leq 1$.

26.1.5

Сформулируйте задачу о максимальном потоке в виде задачи линейного программирования.

26.1.6

У профессора двое детей, которые, к сожалению, терпеть не могут друг друга. Проблема настолько серьезна, что они не только не хотят вместе ходить в школу, но даже отказываются заходить в квартал, в котором в этот день побывал другой. При этом они допускают, что их пути могут пересекаться на углу того или иного квартала. К счастью, и дом профессора, и школа расположены на углах кварталов, однако профессор не уверен, возможно ли отправить обоих детей в одну школу. У профессора есть карта города. Покажите, как сформулировать задачу о возможности отправить детей в одну и ту же школу в виде задачи о максимальном потоке.

26.1.7

Предположим, что в дополнение к пропускным способностям ребер транспортная сеть имеет ***пропускные способности вершин***, т.е. каждая вершина v имеет предел $l(v)$, определяющий, какой величины поток может через нее проходить. Покажите, как преобразовать транспортную сеть $G = (V, E)$ с пропускными способностями вершин в эквивалентную транспортную сеть $G' = (V', E')$ без пропускных способностей вершин, такую, что максимальный поток в G' имеет ту же величину, что и максимальный поток в G. Сколько ребер и вершин входят в G'?

26.2. Метод Форда–Фалкерсона

В данном разделе представлен метод Форда–Фалкерсона для решения задачи о максимальном потоке. Мы называем его методом, а не алгоритмом, поскольку он допускает несколько реализаций с различным временем выполнения. Метод Форда–Фалкерсона базируется на трех важных идеях, которые выходят за рамки данного метода и применяются во многих потоковых алгоритмах и задачах. Это — остаточные сети, увеличивающие пути и разрезы. Данные концепции лежат в основе важной теоремы о максимальном потоке и минимальном разрезе (теорема 26.6), которая определяет значение максимального потока через разрезы транспортной сети. В заключение данного раздела мы предложим одну конкретную реализацию метода Форда–Фалкерсона и проанализируем время ее выполнения.

Метод Форда–Фалкерсона итеративно увеличивает значение потока. Вначале поток обнуляется: $f(u, v) = 0$ для всех $u, v \in V$. На каждой итерации величина потока в G увеличивается посредством поиска "увеличивающего пути" в связанной "остаточной сети" G_f. Зная ребра увеличивающего пути в G_f, мы можем легко идентифицировать конкретные ребра в G, для которых можно изменить поток таким образом, что его величина увеличится. Хотя каждая итерация метода Форда–Фалкерсона увеличивает величину потока, мы увидим, что поток через конкретное ребро G может возрастать или уменьшаться; уменьшение потока через некоторые ребра может быть необходимым для того, чтобы позволить алгоритму переслать больший поток от истока к стоку. Мы многократно увеличиваем

поток до тех пор, пока остаточная сеть не будет иметь ни одного увеличивающего пути. В теореме о максимальном потоке и минимальном разрезе будет показано, что по завершении данного процесса получается максимальный поток.

```
FORD-FULKERSON-METHOD(G, s, t)
1  Инициализация потока f нулевым значением
2  while существует увеличивающий путь p в остаточной сети G_f
3      увеличиваем поток f вдоль пути p
4  return f
```

Чтобы реализовать и проанализировать метод Форда–Фалкерсона, необходимо ввести несколько дополнительных концепций.

Остаточные сети

Интуитивно понятно, что если заданы некоторая транспортная сеть G и поток f, то остаточная сеть G_f — это сеть, состоящая из ребер с пропускными способностями, указывающими, как могут меняться потоки через ребра G. Ребро транспортной сети может пропустить дополнительный поток, равный пропускной способности ребра минус поток, проходящий через это ребро. Если это значение положительно, мы помещаем такое ребро в G_f с "остаточной пропускной способностью" $c_f(u,v) = c(u,v) - f(u,v)$. Дополнительный поток могут пропустить только те ребра в G, которые входят в G_f; ребра (u,v), поток через которые равен их пропускной способности, имеют $c_f(u,v) = 0$ и не входят в G_f.

Однако остаточная сеть G_f может также включать ребра, не входящие в G. Когда алгоритм работает с потоком с целью его увеличения, ему может потребоваться уменьшить поток в некотором конкретном ребре. Чтобы представить возможное уменьшение положительного потока $f(u,v)$ в ребре в G, мы помещаем ребро (v,u) в G_f с остаточной пропускной способностью $c_f(v,u) = f(u,v)$, т.е. ребро, которое может пропустить поток в направлении, обратном к (u,v), не больше потока, идущего по ребру (u,v). Эти обратные ребра в остаточной сети позволяют алгоритму пересылать обратно поток, уже переданный по ребру. Пересылка в обратном направлении эквивалентна *уменьшению* потока в ребре, которое во многих алгоритмах является необходимой операцией.

Говоря более формально, предположим, что у нас есть транспортная сеть $G = (V,E)$ с истоком s и стоком t. Пусть в G имеется поток f, и рассмотрим пару вершин $u, v \in V$. Определим ***остаточную пропускную способность*** $c_f(u,v)$ как

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v)\ , & \text{если } (u,v) \in E\ , \\ f(v,u)\ , & \text{если } (v,u) \in E\ , \\ 0 & \text{в противном случае}\ . \end{cases} \tag{26.2}$$

В силу нашего предположения о том, что из $(u,v) \in E$ вытекает $(v,u) \notin E$, к каждой упорядоченной паре вершин применим только один случай из (26.2).

В качестве примера (26.2), если $c(u,v) = 16$ и $f(u,v) = 11$, мы можем увеличить $f(u,v)$ на величину, не превышающую $c_f(u,v) = 5$ единиц, прежде чем

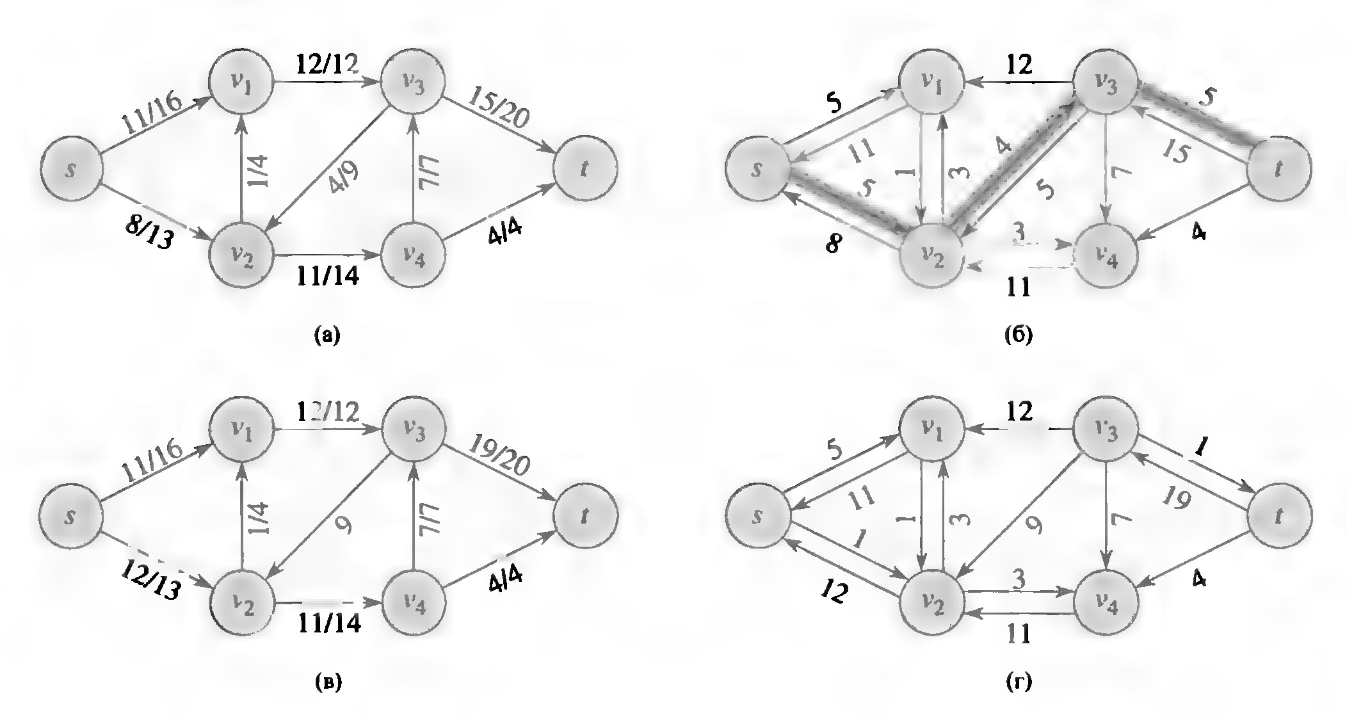


Рис. 26.4. **(а)** Транспортная сеть G и поток f, показанные на рис. 26.1, (б). **(б)** Остаточная сеть G_f с выделенным штриховкой увеличивающим путем p; его остаточная пропускная способность равна $c_f(p) = c_f(v_2, v_3) = 4$. Ребра с остаточной пропускной способностью, равной 0, такие как (v_1, v_3), не показаны (соглашение, которому мы будем следовать в оставшейся части этого раздела). **(в)** Поток в G, полученный в результате увеличения вдоль пути p на его остаточную пропускную способность 4. Ребра, не несущие поток, такие как (v_3, v_2), помечены только пропускной способностью (еще одно соглашение, которому мы будем следовать). **(г)** Остаточная сеть, порожденная потоком в (в).

превысим ограничение пропускной способности ребра (u, v). Мы также хотим позволить алгоритму возвращать до 11 единиц потока назад от v к u, и следовательно, $c_f(v, u) = 11$.

Для заданной транспортной сети $G = (V, E)$ и потока f ***остаточная сеть*** (residual network) G, порожденная потоком f, представляет собой граф $G_f = (V, E_f)$, где

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} \ . \tag{26.3}$$

Иначе говоря, как и отмечалось выше, по каждому ребру остаточной сети, или ***остаточному ребру***, можно направить поток, больший 0. На рис. 26.4, (а) воспроизведены транспортная сеть G и поток f, представленные на рис. 26.1, (б), а на рис. 26.4, (б) показана соответствующая остаточная сеть G_f. Ребра в E_f являются либо ребрами из E, либо обратные им, и, таким образом,

$$|E_f| \le 2\,|E| \ .$$

Заметим, что остаточная сеть G_f подобна транспортной сети с пропускными способностями, задаваемыми c_f. Она не удовлетворяет нашему определению транспортной сети, так как может содержать одновременно и ребро (u, v), и обратное ему ребро (v, u). Помимо этого различия, остаточная сеть обладает всеми свойствами транспортной сети, и мы можем определить поток в остаточной се-

ти как удовлетворяющий определению потока, но по отношению к пропускным способностям c_f сети G_f.

Поток в остаточной сети предоставляет указания по добавлению потока к исходной транспортной сети. Если f представляет собой поток в G, а f' представляет собой поток в соответствующей остаточной сети G_f, определим ***увеличение*** (augmentation) $f \uparrow f'$ потока f на f' как функцию, отображающую $V \times V$ на $\mathbb{R}$, определенную следующим образом:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) \ , & \text{если } (u,v) \in E \ , \\ 0 & \text{в противном случае} \ . \end{cases} \quad (26.4)$$

Интуитивно понятно, что это определение следует из определения остаточной сети. Мы увеличиваем поток в ребре (u,v) на $f'(u,v)$, но уменьшаем его на $f'(v,u)$, поскольку пропускание потока по обратным ребрам в остаточной сети означает уменьшение потока в исходной сети. Пропускание потока по обратному ребру в остаточной сети известно также как ***сокращение*** (cancellation). Например, если мы пересылаем 5 ящиков с хоккейными шайбами из u в v, а 2 ящика — из v в u, то это эквивалентно (конечно, с точки зрения окончательного результата, а не оплаты перевозок) пересылке 3 ящиков из u в v, и ничего — из v в u. Сокращения такого вида являются ключевым моментом любого алгоритма максимального потока.

Лемма 26.1

Пусть $G = (V,E)$ является транспортной сетью с истоком s и стоком t и пусть f представляет собой поток в G. Пусть G_f — остаточная сеть G, порожденная f, и пусть f' — поток в G_f. Тогда функция $f \uparrow f'$, определенная в уравнении (26.4), представляет собой поток в G с величиной $|f \uparrow f'| = |f| + |f'|$.

Доказательство. Сначала убедимся, что $f \uparrow f'$ подчиняется ограничению пропускной способности для каждого ребра в E и сохранению потока в каждой вершине в $V - \{s,t\}$.

В случае ограничения пропускной способности сначала заметим, что если $(u,v) \in E$, то $c_f(v,u) = f(u,v)$. Таким образом, мы имеем $f'(v,u) \leq c_f(v,u) = f(u,v)$, а следовательно

$$\begin{aligned} (f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) \text{ (согласно уравнению (26.4))} \\ &\geq f(u,v) + f'(u,v) - f(u,v) \text{ (поскольку } f'(v,u) \leq f(u,v)) \\ &= f'(u,v) \\ &\geq 0 \ . \end{aligned}$$

Кроме того,

$$
\begin{aligned}
&(f \uparrow f')(u,v) \\
&\qquad = f(u,v) + f'(u,v) - f'(v,u) && \text{(согласно уравнению (26.4))} \\
&\qquad \le f(u,v) + f'(u,v) && \text{(так как потоки неотрицательны)} \\
&\qquad \le f(u,v) + c_f(u,v) && \text{(ограничение пропускной способности)} \\
&\qquad = f(u,v) + c(u,v) - f(u,v) && \text{(определение } c_f\text{)} \\
&\qquad = c(u,v) \ .
\end{aligned}
$$

В случае сохранения потока, поскольку и f, и f' подчиняются свойству сохранения потока, для всех $u \in V - \{s,t\}$ имеем

$$
\begin{aligned}
\sum_{v \in V} (f \uparrow f')(u,v) &= \sum_{v \in V} (f(u,v) + f'(u,v) - f'(v,u)) \\
&= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) \\
&= \sum_{v \in V} f(v,u) + \sum_{v \in V} f'(v,u) - \sum_{v \in V} f'(u,v) \\
&= \sum_{v \in V} (f(v,u) + f'(v,u) - f'(u,v)) \\
&= \sum_{v \in V} (f \uparrow f')(v,u) \ ,
\end{aligned}
$$

где третья строка следует из второй согласно сохранению потока.

Наконец вычислим величину $f \uparrow f'$. Вспомним, что антипараллельные ребра в G (но не в G_f) запрещены, а следовательно, для каждой вершины $v \in V$ мы знаем, что может иметься либо ребро (s,v), либо ребро (v,s), но не оба одновременно. Определим $V_1 = \{v : (s,v) \in E\}$ как множество вершин с ребрами из s, а $V_2 = \{v : (v,s) \in E\}$ как множество вершин с ребрами в s. Мы имеем $V_1 \cup V_2 \subseteq V$ и, поскольку антипараллельные ребра запрещены, $V_1 \cap V_2 = \emptyset$. Теперь вычислим

$$
\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s,v) - \sum_{v \in V} (f \uparrow f')(v,s) \\
&= \sum_{v \in V_1} (f \uparrow f')(s,v) - \sum_{v \in V_2} (f \uparrow f')(v,s) \ , \qquad (26.5)
\end{aligned}
$$

где вторая строка следует из того, что $(f \uparrow f')(w,x)$ равно 0, если $(w,x) \notin E$. Теперь применим к уравнению (26.5) определение $f \uparrow f'$, а затем переупорядочим

и сгруппируем члены, чтобы получить

$$\begin{aligned}
&|f \uparrow f'| \\
&= \sum_{v \in V_1} (f(s,v) + f'(s,v) - f'(v,s)) - \sum_{v \in V_2} (f(v,s) + f'(v,s) - f'(s,v)) \\
&= \sum_{v \in V_1} f(s,v) + \sum_{v \in V_1} f'(s,v) - \sum_{v \in V_1} f'(v,s) \\
&\qquad - \sum_{v \in V_2} f(v,s) - \sum_{v \in V_2} f'(v,s) + \sum_{v \in V_2} f'(s,v) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) \\
&\qquad + \sum_{v \in V_1} f'(s,v) + \sum_{v \in V_2} f'(s,v) - \sum_{v \in V_1} f'(v,s) - \sum_{v \in V_2} f'(v,s) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1 \cup V_2} f'(s,v) - \sum_{v \in V_1 \cup V_2} f'(v,s) \; . \qquad (26.6)
\end{aligned}$$

В уравнении (26.6) можно распространить все четыре суммы на суммирование по всему множеству V, поскольку все дополнительные члены в этом случае имеют значения 0 (в упр. 26.2.1 это требуется строго доказать). Таким образом, мы имеем

$$\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s) \qquad (26.7) \\
&= |f| + |f'| \; .
\end{aligned}$$

■

Увеличивающие пути

Для заданных транспортной сети $G = (V, E)$ и потока f ***увеличивающим путем*** (augmenting path) p является простой путь из s в t в остаточной сети G_f. Согласно определению остаточной сети мы можем увеличить поток в ребре (u, v) увеличивающего пути до $c_f(u, v)$ без нарушения ограничения пропускной способности соответствующего ребра в исходной сети.

Выделенный путь на рис. 26.4, (б) является увеличивающим путем. Рассматривая представленную на рисунке остаточную сеть G_f как некоторую транспортную сеть, можно увеличивать поток вдоль каждого ребра данного пути вплоть до четырех единиц, не нарушая ограничений пропускной способности, поскольку наименьшая остаточная пропускная способность на данном пути составляет $c_f(v_2, v_3) = 4$. Максимальная величина, на которую можно увеличить поток в каждом ребре увеличивающего пути p, называется ***остаточной пропускной способностью*** (residual capacity) пути p и задается формулой

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ принадлежит } p\} \; .$$

Следующая лемма, доказательство которой предлагается провести в качестве упр. 26.2.7, более строго формулирует приведенные выше рассуждения.

Лемма 26.2

Пусть $G = (V, E)$ является транспортной сетью, а f представляет собой поток в G, и пусть p является увеличивающим путем в G_f. Определим функцию $f_p : V \times V \to \mathbb{R}$ как

$$f_p(u,v) = \begin{cases} c_f(p), & \text{если } (u,v) \text{ принадлежит } p, \\ 0 & \text{в противном случае}. \end{cases} \tag{26.8}$$

Тогда f_p является потоком в G_f с величиной $|f_p| = c_f(p) > 0$. ■

Вытекающее из данной леммы следствие показывает, что если увеличить f на f_p, то можно получить новый поток в G, величина которого ближе к максимальной. На рис. 26.4, (в) показан результат увеличения потока f, представленного на рис. 26.4, (а), на поток f_p, показанный на рис. 26.4, (б), а на рис. 26.4, (г) показана полученная остаточная сеть.

Следствие 26.3

Пусть $G = (V, E)$ представляет собой транспортную сеть, а f является потоком в G, и пусть p представляет собой увеличивающий путь в G_f. Пусть также f_p определен, как в уравнении (26.8), и предположим, что мы увеличиваем f на f_p. Тогда функция $f \uparrow f_p$ является потоком в G с величиной $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Доказательство. Непосредственно вытекает из лемм 26.1 и 26.2. ■

Разрезы транспортных сетей

В методе Форда–Фалкерсона проводится многократное увеличение потока вдоль увеличивающих путей до тех пор, пока не будет найден максимальный поток. Откуда нам известно, что по завершении алгоритма мы действительно найдем максимальный поток? В теореме о максимальном потоке и минимальном разрезе, которую мы вскоре докажем, утверждается, что поток является максимальным тогда и только тогда, когда его остаточная сеть не содержит увеличивающих путей. Однако для доказательства данной теоремы необходимо ввести понятие разреза транспортной сети.

Разрезом (cut) (S, T) транспортной сети $G = (V, E)$ называется разбиение множества вершин V на множества S и $T = V - S$, такие, что $s \in S$, а $t \in T$. (Это определение аналогично определению разреза, которое использовалось применительно к минимальным остовным деревьям в главе 23, однако здесь речь идет о разрезе в ориентированном графе, а не в неориентированном, и мы требуем, чтобы $s \in S$, а $t \in T$.) Если f — поток, то ***чистый поток*** (net flow) $f(S, T)$ через разрез (S, T) определяется как

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) . \tag{26.9}$$

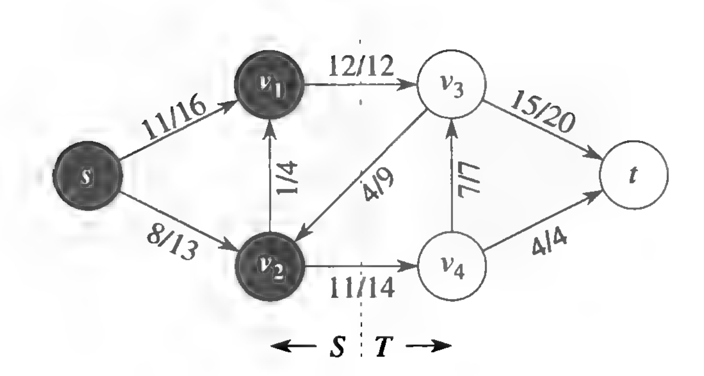


Рис. 26.5. Разрез (S,T) в транспортной сети, показанной на рис. 26.1, (б), где $S = \{s, v_1, v_2\}$, а $T = \{v_3, v_4, t\}$. Вершины в S показаны черными, а вершины в T — белыми. Чистый поток через разрез (S,T) равен $f(S,T) = 19$, а пропускная способность составляет $c(S,T) = 26$.

Пропускной способностью (capacity) разреза (S,T) является

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) . \tag{26.10}$$

Минимальным разрезом (minimum cut) сети является разрез, пропускная способность которого среди всех разрезов сети минимальна.

Асимметрия между определениями потока и пропускной способности разреза является преднамеренной и существенной. В случае пропускной способности мы учитываем только пропускные способности ребер, идущих из S в T, игнорируя ребра, идущие в обратном направлении. Что касается потока, то мы рассматриваем поток из S в T минус поток, идущий в обратном направлении, из T в S. Причина этой разницы в определениях станет ясной позже в этом разделе.

На рис. 26.5 показан разрез $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ транспортной сети, представленной на рис. 26.1, (б). Чистый поток через данный разрез равен

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\ &= 19 , \end{aligned}$$

а пропускная способность данного разреза составляет

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

Следующая лемма показывает, что для заданного потока f чистый поток через любой разрез одинаков и равен величине потока $|f|$.

Лемма 26.4

Пусть f представляет собой поток в транспортной сети G со стоком s и истоком t и пусть (S,T) — произвольный разрез G. Тогда чистый поток через разрез (S,T) равен $f(S,T) = |f|$.

Доказательство. Условие сохранения потока для любого узла $u \in V - \{s,t\}$ можно переписать как

$$\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = 0 . \qquad (26.11)$$

С учетом определения $|f|$ из уравнения (26.1), добавляя равную 0 левую часть уравнения (26.11), и суммируя по всем вершинам в $S - \{s\}$, получим

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right) .$$

Расширение правой суммы и перегруппировка членов дает

$$\begin{aligned}
|f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u,v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v,u) \\
&= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S - \{s\}} f(u,v) \right) - \sum_{v \in V} \left(f(v,s) + \sum_{u \in S - \{s\}} f(v,u) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u) .
\end{aligned}$$

Поскольку $V = S \cup T$ и $S \cap T = \emptyset$, мы можем разбить каждое суммирование по V на суммы по S и T и получить

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&\quad + \left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right) .
\end{aligned}$$

Две суммы в скобках на самом деле одинаковы, поскольку для всех вершин $x, y \in S$ член $f(x,y)$ в каждую сумму входит по одному разу. Следовательно, эти суммы сокращаются, и мы имеем

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= f(S,T) .
\end{aligned}$$

■

Следствие леммы 26.4 показывает, как пропускные способности разрезов можно использовать для определения границы величины потока.

Следствие 26.5

Величина любого потока f в транспортной сети G ограничена сверху пропускной способностью произвольного разреза G.

Доказательство. Пусть (S,T) представляет собой произвольный разрез G и пусть f является произвольным потоком. Согласно лемме 26.4 и ограничению пропускной способности

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ &\leq \sum_{u\in S}\sum_{v\in T} f(u,v) \\ &\leq \sum_{u\in S}\sum_{v\in T} c(u,v) \\ &= c(S,T) \;. \end{aligned}$$ ■

Непосредственно из следствия 26.5 вытекает, что величина максимального потока в сети ограничена сверху пропускной способностью минимального разреза. Сейчас мы сформулируем и докажем важную теорему о максимальном потоке и минимальном разрезе, в которой утверждается, что значение максимального потока равно пропускной способности минимального разреза.

Теорема 26.6 (О максимальном потоке и минимальном разрезе)

Если f представляет собой поток в транспортной сети $G = (V,E)$ с истоком s и стоком t, то следующие утверждения эквивалентны.

1. f является максимальным потоком в G.
2. Остаточная сеть G_f не содержит увеличивающих путей.
3. $|f| = c(S,T)$ для некоторого разреза (S,T) транспортной сети G.

Доказательство. (1) $\Rightarrow$ (2): предположим противное: пусть f является максимальным потоком в G, но G_f содержит увеличивающий путь p. Тогда, согласно следствию 26.3, поток, полученный путем увеличения потока f на поток f_p, где f_p задается уравнением (26.8), представляет собой поток в G с величиной, строго большей, чем $|f|$, что противоречит предположению о том, что f является максимальным потоком.

(2) $\Rightarrow$ (3): предположим, что G_f не содержит увеличивающего пути, т.е. что в G_f нет пути из s в t. Определим

$$S = \{v \in V : \text{в } G_f \text{ имеется путь из } s \text{ в } v\}$$

и $T = V - S$. Разбиение (S,T) является разрезом: $s \in S$ выполняется тривиально, а $t \notin S$, поскольку в G_f нет пути из s в t. Теперь рассмотрим пару вершин $u \in S$ и $v \in T$. Если $(u,v) \in E$, должно выполняться $f(u,v) = c(u,v)$, поскольку

в противном случае $(u, v) \in E_f$, что помещало бы вершину v в множество S. Если $(v, u) \in E$, должно выполняться $f(v, u) = 0$, поскольку в противном случае значение $c_f(u, v) = f(v, u)$ должно было бы быть положительным и мы должны были бы иметь $(u, v) \in E_f$, так что v должно было бы находиться в S. Конечно, если ни (u, v), ни (v, u) не находятся в E, то $f(u, v) = f(v, u) = 0$. Таким образом, имеем

$$\begin{aligned} f(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\ &= \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{v\in T}\sum_{u\in S} 0 \\ &= c(S,T) \ . \end{aligned}$$

Следовательно, согласно лемме 26.4 $|f| = f(S,T) = c(S,T)$.

(3) $\Rightarrow$ (1): согласно следствию 26.5 $|f| \leq c(S,T)$ для всех разрезов (S,T). Таким образом, из условия $|f| = c(S,T)$ вытекает, что поток f является максимальным потоком. ■

Базовый алгоритм Форда–Фалкерсона

При выполнении каждой итерации метода Форда–Фалкерсона мы находим *некоторый* увеличивающий путь p и используем p для того, чтобы изменять поток f. Как предлагается леммой 26.2 и следствием 26.3, мы заменяем f на $f \uparrow f_p$, получая новый поток, величина которого равна $|f| + |f_p|$. Приведенная далее реализация данного метода вычисляет максимальный поток в транспортной сети $G = (V, E)$ путем обновления атрибута потока $(u, v).f$ каждого ребра $(u, v) \in E$[1]. Если $(u, v) \notin E$, неявно предполагается, что $(u, v).f = 0$. Мы также считаем, что вместе с транспортной сетью задаются пропускные способности $c(u, v)$ и что $c(u, v) = 0$, если $(u, v) \notin E$. Остаточная пропускная способность $c_f(u, v)$ вычисляется по формуле (26.2). Выражение $c_f(p)$ в коде процедуры является просто временной переменной, в которой хранится остаточная пропускная способность пути p.

```
FORD-FULKERSON(G, s, t)
for каждого ребра (u, v) ∈ G.E
    (u, v).f = 0
while существует путь p из s в t в остаточной сети G_f
    c_f(p) = min {c_f(u, v) : (u, v) содержится в p}
    for каждого ребра (u, v) в p
        if (u, v) ∈ E
            (u, v).f = (u, v).f + c_f(p)
        else (v, u).f = (v, u).f − c_f(p)
```

[1]Вспомним из раздела 22.1, что мы представляем атрибут f для ребра (u, v) с помощью того же стиля обозначений — $(u, v).f$, — что и в случае атрибутов любого другого объекта.

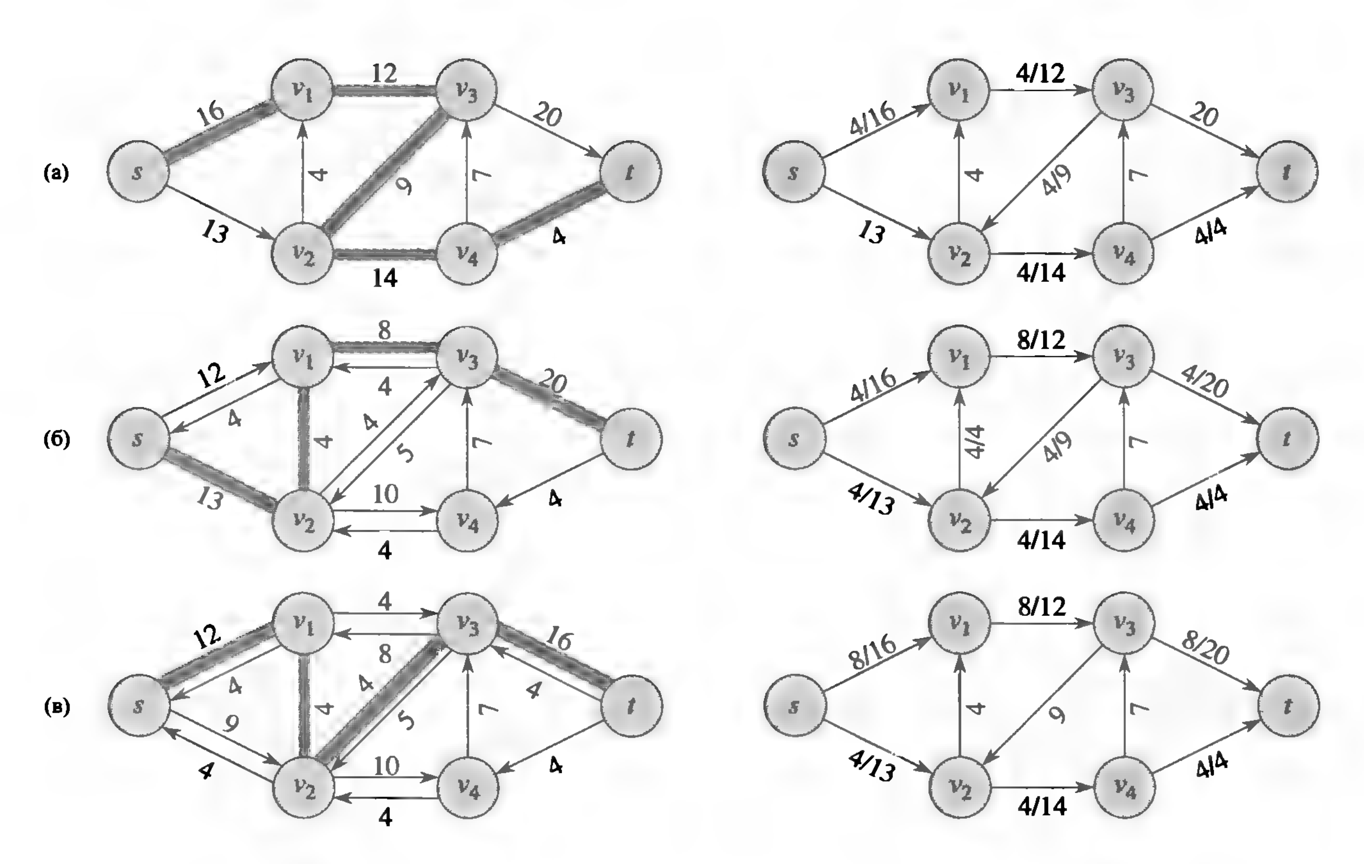


Рис. 26.6. Работа базового алгоритма Форда–Фалкерсона. **(а)–(д)** Последовательные итерации цикла **while**. Слева в каждой части показана остаточная сеть G_f из строки 3 с выделенным увеличивающим путем p. Справа показан новый поток f, который является результатом увеличения f на f_p. Остаточная сеть в (а) представляет собой входную сеть G.

Процедура FORD-FULKERSON является расширением приведенного ранее псевдокода FORD-FULKERSON-METHOD. На рис. 26.6 показаны результаты каждой итерации при тестовом выполнении. Строки 1 и 2 инициализируют поток f значением 0. В цикле **while** в строках 3–8 выполняется неоднократный поиск увеличивающего пути p в G_f и увеличение потока f вдоль пути p увеличивается на остаточную пропускную способность $c_f(p)$. Каждое остаточное ребро в пути p является либо ребром исходной сети, либо обратным к ребру в исходной сети. В строках 6–8 выполняется обновление потока, соответствующее каждому случаю, путем добавления потока, если остаточное ребро является ребром исходной сети, и вычитания в противном случае. Когда увеличивающих путей больше нет, поток f является максимальным.

Анализ метода Форда–Фалкерсона

Время выполнения процедуры FORD-FULKERSON зависит от того, как именно выполняется поиск увеличивающего пути p в строке 3. При неудачном выборе метода поиска алгоритм может даже не завершиться: величина потока будет последовательно увеличиваться, но она не обязательно будет сходиться к макси-

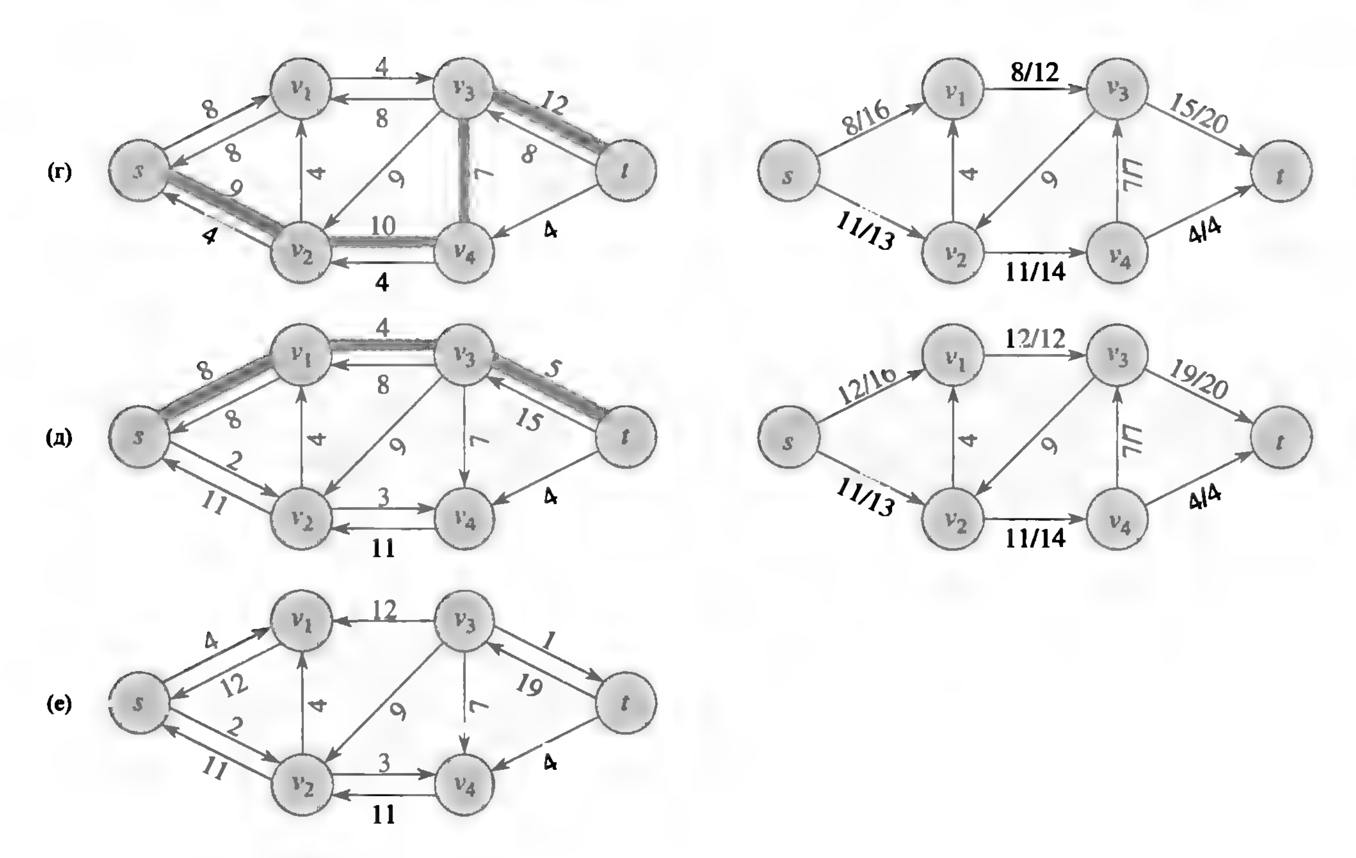


Рис. 26.6 (продолжение). **(е)** Остаточная сеть при последней проверке цикла **while**. В ней нет увеличивающих путей, так что поток f, показанный в части (д), является максимальным. Величина найденного максимального потока равна 23.

мальному значению потока[2]. Если увеличивающий путь выбирается с использованием поиска в ширину (который мы рассматривали в разделе 22.2), алгоритм выполняется за полиномиальное время. Прежде чем доказать этот результат, получим простую границу времени выполнения для случая, когда увеличивающий путь выбирается произвольным образом, а все значения пропускных способностей являются целыми числами.

На практике задача поиска максимального потока часто возникает в целочисленной постановке. Если пропускные способности являются рациональными числами, можно использовать соответствующее масштабирование, которое сделает их целыми. Если обозначить максимальный поток в такой трансформированной сети как f^*, то в случае непосредственной реализации процедуры FORD-FULKERSON цикл **while** в строках 3–8 выполняется не более $|f^*|$ раз, поскольку величина потока за каждую итерацию увеличивается по крайней мере на одну единицу.

Цикл **while** будет выполняться эффективно, если реализовать транспортную сеть $G = (V, E)$ с помощью правильно выбранной структуры данных и искать увеличивающий путь с помощью алгоритма с линейным временем работы. Предположим, что мы поддерживаем структуру данных, соответствующую ориентированному графу $G' = (V, E')$, где $E' = \{(u, v) : (u, v) \in E$ или $(v, u) \in E\}$. Ребра

[2]Метод Форда–Фалкерсона может работать бесконечно, только если значения пропускной способности ребер являются иррациональными числами.

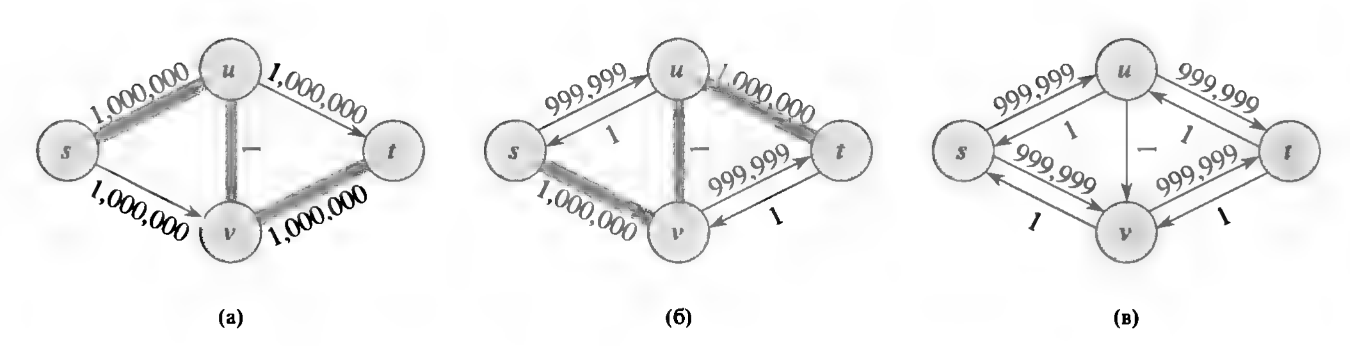


Рис. 26.7. **(а)** Транспортная сеть, для обработки которой алгоритму FORD-FULKERSON может потребоваться время $\Theta(E\,|f^*|)$, где f^* представляет собой максимальный поток, который в данном случае имеет величину $|f^*| = 2\,000\,000$. Штриховкой выделен увеличивающий путь с остаточной пропускной способностью 1. **(б)** Полученная остаточная сеть с другим увеличивающим путем с остаточной пропускной способностью 1. **(в)** Полученная в результате остаточная сеть.

сети G являются также ребрами графа G', поэтому в такой структуре данных можно довольно легко хранить пропускные способности и потоки. Для данного потока f в G ребра остаточной сети G_f состоят из всех ребер (u, v) графа G', таких, что $c_f(u, v) > 0$, где c_f удовлетворяет уравнению (26.2). Следовательно, время поиска пути в остаточной сети составляет $O(V + E') = O(E)$, если используется либо поиск в глубину, либо поиск в ширину. Таким образом, каждая итерация цикла **while** занимает время $O(E)$, что вместе с инициализацией в строках 1 и 2 делает общее время выполнения алгоритма FORD-FULKERSON равным $O(E\,|f^*|)$.

Когда значения пропускных способностей являются целыми числами и оптимальное значение потока $|f^*|$ невелико, время выполнения алгоритма Форда–Фалкерсона достаточно неплохое. Но на рис. 26.7, (а) показан пример того, что может произойти в простой транспортной сети с бо́льшим значением $|f^*|$. Величина максимального потока в данной сети равна 2 000 000: 1 000 000 единиц потока идет по пути $s \to u \to t$, а другие 1 000 000 единиц идут по пути $s \to v \to t$. Если первым увеличивающим путем, найденным процедурой FORD-FULKERSON, является путь $s \to u \to v \to t$, как показано на рис. 26.7, (а), поток после первой итерации имеет значение 1. Полученная остаточная сеть показана на рис. 26.7, (б). Если в ходе выполнения второй итерации будет найден увеличивающий путь $s \to v \to u \to t$, как показано на рис. 26.7, (б), поток станет равным 2. На рис. 26.7, (в) показана соответствующая остаточная сеть. Можно продолжать процедуру, выбирая увеличивающий путь $s \to u \to v \to t$ в нечетных итерациях и $s \to v \to u \to t$ в четных. В таком случае нам придется выполнить 2 000 000 увеличений, при этом величина потока на каждом шаге увеличивается всего на 1.

Алгоритм Эдмондса–Карпа

Можно улучшить временну́ю границу алгоритма FORD-FULKERSON, если реализовать вычисление увеличивающего пути p в строке 3 как поиск в ширину, т.е. если в качестве увеличивающего пути выбрать *кратчайший* путь из s в t в остаточной сети, где каждое ребро имеет единичную длину (вес). Такая реализация метода Форда–Фалкерсона называется ***алгоритмом Эдмондса–Карпа***

(Edmonds–Karp algorithm). Докажем, что время выполнения алгоритма Эдмондса–Карпа составляет $O(VE^2)$.

Анализ зависит от расстояний между вершинами остаточной сети G_f. В следующей лемме длина кратчайшего пути из вершины u в v в остаточной сети G_f, где каждое ребро имеет единичную длину, обозначена как $\delta_f(u, v)$.

Лемма 26.7

Если применить алгоритм Эдмондса–Карпа к транспортной сети $G = (V, E)$ с истоком s и стоком t, то для всех вершин $v \in V - \{s, t\}$ длина кратчайшего пути $\delta_f(s, v)$ в остаточной сети G_f монотонно увеличивается с каждым увеличением потока.

Доказательство. Предположим, что для некоторой вершины $v \in V - \{s, t\}$ существует такое увеличение потока, которое приводит к уменьшению длины кратчайшего пути из s в v, и покажем, что это предположение приведет нас к противоречию. Пусть f — поток, который был непосредственно перед первым увеличением, приведшим к уменьшению длины некоего кратчайшего пути, а f' — поток сразу после этого увеличения. Пусть v — вершина с минимальной длиной кратчайшего пути $\delta_{f'}(s, v)$, которая уменьшилась в результате увеличения потока, так что $\delta_{f'}(s, v) < \delta_f(s, v)$. Пусть $p = s \leadsto u \to v$ — кратчайший путь из s в v в $G_{f'}$, такой, что $(u, v) \in E_{f'}$ и

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 . \qquad (26.12)$$

Исходя из того, как мы выбирали v, можно утверждать, что длина пути до вершины u из истока s не уменьшилась, т.е.

$$\delta_{f'}(s, u) \geq \delta_f(s, u) . \qquad (26.13)$$

Мы утверждаем, что $(u, v) \notin E_f$. Почему? Если мы имели $(u, v) \in E_f$, то должны были также выполняться соотношения

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(лемма 24.10, неравенство треугольника)} \\ &\leq \delta_{f'}(s, u) + 1 && \text{(согласно неравенству (26.13))} \\ &= \delta_{f'}(s, v) && \text{(согласно уравнению (26.12))} , \end{aligned}$$

что противоречит нашему предположению о том, что $\delta_{f'}(s, v) < \delta_f(s, v)$.

Как же может получиться $(u, v) \notin E_f$ и $(u, v) \in E_{f'}$? Увеличение должно привести к возрастанию потока из v в u. Алгоритм Эдмондса–Карпа всегда увеличивает поток вдоль кратчайших путей, поэтому последним ребром кратчайшего пути из s в u в G_f является ребро (v, u). Следовательно,

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 && \text{(согласно неравенству (26.13))} \\ &= \delta_{f'}(s, v) - 2 && \text{(согласно уравнению (26.12))} , \end{aligned}$$

что противоречит нашему предположению о том, что $\delta_{f'}(s,v) < \delta_f(s,v)$. Мы заключаем, что наше предположение о существовании такой вершины v неверное. ■

В следующей теореме устанавливается верхний предел количества итераций алгоритма Эдмондса–Карпа.

Теорема 26.8

Если алгоритм Эдмондса–Карпа выполняется для некоторой транспортной сети $G = (V, E)$ с истоком s и стоком t, то общее число увеличений потока, выполняемых данным алгоритмом, составляет $O(VE)$.

Доказательство. Назовем ребро (u, v) остаточной сети G_f ***критическим*** (critical) для увеличивающего пути p, если остаточная пропускная способность p равна остаточной пропускной способности ребра (u, v), т.е. если $c_f(p) = c_f(u, v)$. После увеличения потока вдоль увеличивающего пути все критические ребра этого пути исчезают из остаточной сети. Кроме того, по крайней мере одно ребро любого увеличивающего пути должно быть критическим. Теперь покажем, что каждое из $|E|$ ребер может становиться критическим не более $|V|/2$ раз.

Пусть u и v являются вершинами из множества вершин V, соединенными некоторым ребром из множества E. Поскольку увеличивающие пути являются кратчайшими путями, то, когда ребро (u, v) становится критическим в первый раз, справедливо равенство

$$\delta_f(s,v) = \delta_f(s,u) + 1 .$$

После того как поток увеличен, ребро (u, v) исчезает из остаточной сети. Оно не может появиться в другом увеличивающем пути, пока не будет уменьшен поток из u в v, а это может произойти только в том случае, если на некотором увеличивающем пути встретится ребро (v, u). Если в этот момент в сети G поток представляет собой f', то справедливо следующее равенство

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 .$$

Поскольку $\delta_f(s,v) \le \delta_{f'}(s,v)$, согласно лемме 26.7, мы имеем

$$\begin{aligned} \delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\ge \delta_f(s,v) + 1 \\ &= \delta_f(s,u) + 2 . \end{aligned}$$

Следовательно, за время, прошедшее с момента, когда ребро (u, v) стало критическим, до момента, когда оно станет критическим в следующий раз, расстояние до u от истока увеличивается не менее чем на 2. Расстояние до u от истока в начальный момент было не меньше 0. Среди промежуточных вершин на кратчайшем пути из s в u не могут находиться s, u или t (поскольку наличие ребра (u, v) в увеличивающем пути влечет за собой $u \ne t$). Следовательно, к тому

моменту, когда вершина u станет недостижимой из истока (если такое произойдет), расстояние до нее будет не более $|V| - 2$. Таким образом, ребро (u, v) может стать критическим не более чем еще $(|V| - 2)/2 = |V|/2 - 1$ раз, т.е. всего не более $|V|/2$ раз. Поскольку в остаточном графе имеется $O(E)$ пар вершин, которые могут быть соединены ребрами в остаточной сети, общее количество критических ребер в ходе выполнения алгоритма Эдмондса–Карпа равно $O(VE)$. Каждый увеличивающий путь содержит по крайней мере одно критическое ребро, а значит, теорема доказана. ■

Поскольку, если использовать поиск в ширину, можно выполнять каждую итерацию процедуры FORD-FULKERSON за время $O(E)$, общее время работы алгоритма Эдмондса–Карпа оказывается равным $O(VE^2)$. Мы покажем, что алгоритмы проталкивания предпотока позволяют достичь еще лучших результатов. На основе алгоритма из раздела 26.4 построен метод, который позволяет достичь времени выполнения $O(V^2E)$; этот метод является основой алгоритма со временем выполнения $O(V^3)$, рассматриваемого в разделе 26.5.

Упражнения

26.2.1

Докажите, что сумма в уравнении (26.6) равна сумме в уравнении (26.7).

26.2.2

Чему равен поток через разрез $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ на рис. 26.1, (б)? Какова пропускная способность этого разреза?

26.2.3

Продемонстрируйте выполнение алгоритма Эдмондса–Карпа на примере транспортной сети, представленной на рис. 26.1, (а).

26.2.4

Укажите на рис. 26.6 минимальный разрез, соответствующий показанному максимальному потоку. Какой из показанных в примере увеличивающих путей приводит к сокращению потока?

26.2.5

Вспомним предложенную в разделе 26.1 конструкцию, которая преобразует транспортную сеть с несколькими истоками и несколькими стоками в сеть с одним истоком и одним стоком путем добавления ребер с бесконечной пропускной способностью. Докажите, что любой поток в полученной сети имеет конечную величину, если ребра исходной сети с множественными истоками и стоками имеют конечную пропускную способность.

26.2.6

Предположим, что каждый исток s_i в задаче с множественными истоками и стоками производит ровно p_i единиц потока, так что $\sum_{v \in V} f(s_i, v) = p_i$. Предположим также, что каждый сток t_j потребляет ровно q_j единиц, так что $\sum_{v \in V} f(v, t_j) = q_j$,

где $\sum_i p_i = \sum_j q_j$. Покажите, как преобразовать данную задачу поиска потока f, удовлетворяющего указанным дополнительным ограничениям, в задачу поиска максимального потока в транспортной сети с одним истоком и одним стоком.

26.2.7

Докажите лемму 26.2.

26.2.8

Предположим, что мы переопределили остаточную сеть, запретив ребра, входящие в s. Докажите, что процедура FORD-FULKERSON все равно будет корректно вычислять максимальный поток.

26.2.9

Предположим, что и f, и f' являются потоками в сети G и что мы вычисляем поток $f \uparrow f'$. Будет ли увеличенный поток удовлетворять свойству сохранения потока? А ограничению пропускной способности?

26.2.10

Покажите, как найти максимальный поток в сети $G = (V, E)$ путем последовательности не более чем из $|E|$ увеличивающих путей. (*Указание:* определите пути *после* нахождения максимального потока.)

26.2.11

Реберной связностью (edge connectivity) неориентированного графа называется минимальное число ребер k, которые необходимо удалить, чтобы разъединить граф. Например, реберная связность дерева равна 1, а реберная связность циклической цепи вершин равна 2. Покажите, как определить реберную связность неориентированного графа $G = (V, E)$ с помощью алгоритма максимального потока не более чем для $|V|$ транспортных сетей, каждая из которых содержит $O(V)$ вершин и $O(E)$ ребер.

26.2.12

Предположим, что имеется транспортная сеть G и что в G имеются ребра, входящие в исток s. Пусть f представляет собой поток в G, в котором одно из входящих в исток ребер (v, s) имеет $f(v, s) = 1$. Докажите, что должен существовать другой поток f' с $f'(v, s) = 0$, такой, что $|f| = |f'|$. Разработайте алгоритм со временем работы $O(E)$, который вычисляет f' по данному f, в предположении, что все пропускные способности ребер являются целыми числами.

26.2.13

Предположим, что вам требуется найти среди всех минимальных разрезов транспортной сети G с целочисленными пропускными способностями тот, который содержит наименьшее количество ребер. Покажите, как изменить пропускные способности G, чтобы создать новую транспортная сеть G', в которой любой минимальный разрез в G' является минимальным разрезом с наименьшим количеством ребер в G.

26.3. Максимальное паросочетание

Некоторые комбинаторные задачи можно легко свести к задачам поиска максимального потока. Одной из таких задач является задача определения максимального потока в сети с несколькими истоками и стоками, описанная в разделе 26.1. Существуют другие комбинаторные задачи, которые, на первый взгляд, имеют мало общего с транспортными сетями, однако могут быть сведены к задачам поиска максимального потока. В данном разделе рассматривается одна из подобных задач: поиск максимального паросочетания в двудольном графе. Чтобы решить данную задачу, воспользуемся свойством полноты, обеспечиваемым методом Форда–Фалкерсона. Мы также покажем, что с помощью метода Форда–Фалкерсона можно решить задачу поиска максимального паросочетания в двудольном графе $G = (V, E)$ за время $O(VE)$.

Задача поиска максимального паросочетания в двудольном графе

Пусть дан неориентированный граф $G = (V, E)$. ***Паросочетанием*** (matching) называется подмножество ребер $M \subseteq E$, такое, что для всех вершин $v \in V$ в M содержится не более одного ребра, инцидентного v. Мы говорим, что вершина $v \in V$ является ***связанной*** (matched) паросочетанием M, если в M есть ребро, инцидентное v; в противном случае вершина v называется ***открытой*** (unmatched). ***Максимальным паросочетанием*** называется паросочетание максимальной мощности, т.е. такое паросочетание M, что для любого паросочетания M' справедливо соотношение $|M| \geq |M'|$. В данном разделе мы ограничимся рассмотрением задачи поиска максимальных паросочетаний в двудольных графах, т.е. в графах, множество вершин которых можно разбить на два подмножества $V = L \cup R$, где L и R не пересекаются и все ребра в E проходят между L и R. Далее мы предполагаем, что каждая вершина в V имеет по крайней мере одно инцидентное ребро. Понятие паросочетания проиллюстрировано на рис. 26.8.

Задача поиска максимального паросочетания в двудольном графе имеет множество практических приложений. В качестве примера можно рассмотреть паросочетание множества машин L и множества задач R, которые должны выполняться одновременно. Наличие в E ребра (u, v) означает, что машина $u \in L$ может выполнять задачу $v \in R$. Максимальное паросочетание обеспечивает максимальную загрузку машин.

Поиск максимального паросочетания в двудольном графе

С помощью метода Форда–Фалкерсона можно найти максимальное паросочетание в неориентированном двудольном графе $G = (V, E)$ за время, полиномиально зависящее от $|V|$ и $|E|$. Фокус заключается в том, чтобы построить транспортную сеть, потоки в которой соответствуют паросочетаниям, как показано на рис. 26.8, (в). Определим для заданного двудольного графа G ***соответствующую транспортную сеть*** $G' = (V', E')$ следующим образом. Возьмем в качестве истока s и стока t новые вершины, не входящие в V, и положим $V' = V \cup \{s, t\}$.

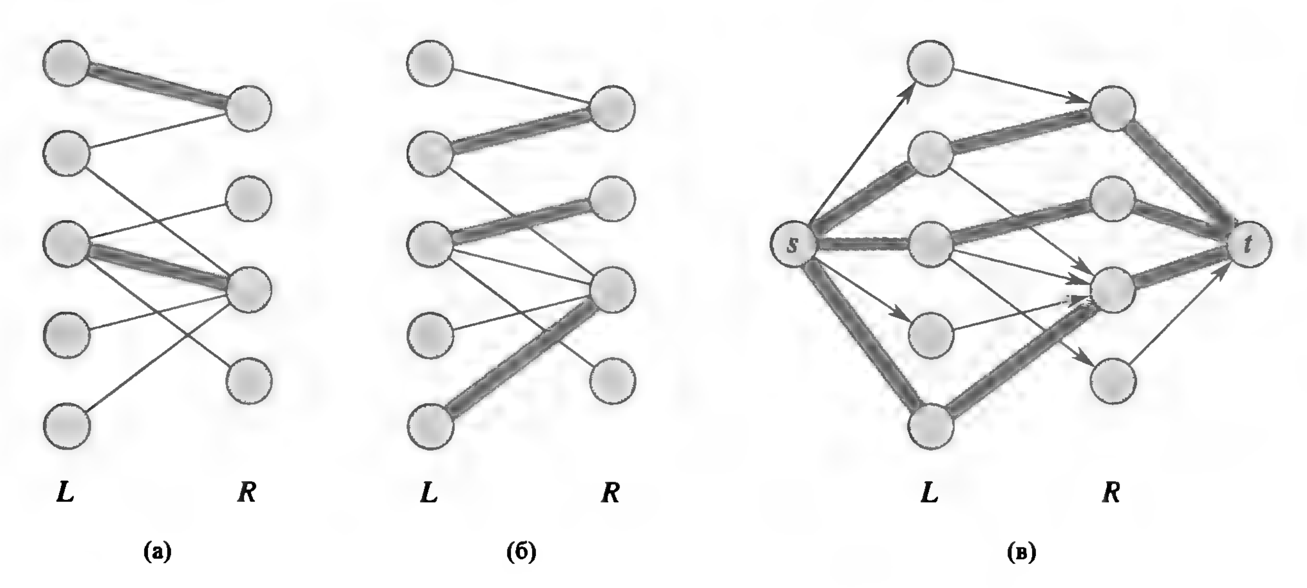


Рис. 26.8. Двудольный граф $G = (V, E)$ с разбиением вершин $V = L \cup R$. **(а)** Паросочетание с мощностью 2 (ребра выделены штриховкой). **(б)** Максимальное паросочетание с мощностью 3. **(в)** Соответствующая транспортная сеть G' с показанным максимальным потоком. Каждое ребро имеет единичную пропускную способность. Через заштрихованные ребра идет поток величиной 1, во всех остальных ребрах потока нет. Заштрихованные ребра из L в R соответствуют заштрихованным ребрам в максимальном паросочетании в части (б).

Если разбиение вершин в графе G представляет собой $V = L \cup R$, ориентированными ребрами G' являются ребра E, направленные из L в R, а также $|V|$ новых ориентированных ребер

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\} .$$

Чтобы завершить построение, присвоим каждому ребру E' единичную пропускную способность. Поскольку каждая вершина из множества вершин V имеет по крайней мере одно инцидентное ребро, $|E| \geq |V|/2$. Таким образом, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, так что $|E'| = \Theta(E)$.

Следующая лемма показывает, что паросочетание в G непосредственно соответствует потоку в соответствующей транспортной сети G'. Поток f в транспортной сети $G = (V, E)$ называется ***целочисленным*** (integer-valued), если значения $f(u, v)$ целые для всех $(u, v) \in V \times V$.

Лемма 26.9

Пусть $G = (V, E)$ является двудольным графом с разбиением вершин $V = L \cup R$ и пусть $G' = (V', E')$ представляет собой соответствующую ему транспортную сеть. Если M является паросочетанием в G, то существует целочисленный поток f в G', величина которого — $|f| = |M|$. Справедливо и обратное утверждение: если f представляет собой целочисленный поток в G', то в G существует паросочетание M с мощностью $|M| = |f|$.

Доказательство. Покажем сначала, что паросочетанию M в графе G соответствует некоторый целочисленный поток f в сети G'. Определим f следующим образом. Если $(u, v) \in M$, то $f(s, u) = f(u, v) = f(v, t) = 1$. Для всех остальных

ребер $(u, v) \in E'$ определим $f(u, v) = 0$. Нетрудно убедиться, что f удовлетворяет ограничению пропускной способности и сохранению потока.

Интуитивно понятно, что каждое ребро $(u, v) \in M$ соответствует единице потока в G', проходящего по пути $s \to u \to v \to t$. Кроме того, пути, порожденные ребрами из M, представляют собой непересекающиеся множества вершин, не считая s и t.

Чистый поток через разрез $(L \cup \{s\}, R \cup \{t\})$ равен $|M|$; следовательно, согласно лемме 26.4 величина потока равна $|f| = |M|$.

Чтобы доказать обратное, предположим, что f — некоторый целочисленный поток в G', и пусть

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{и } f(u, v) > 0\} .$$

Каждая вершина $u \in L$ имеет только одно входящее ребро, а именно — (s, u), и его пропускная способность равна 1. Следовательно, в каждую вершину $u \in L$ входит не более одной единицы положительного потока, и если она действительно входит, то из нее должна также выходить одна единица положительного потока согласно свойству сохранения потока. Более того, поскольку f — целочисленный поток, для каждой вершины $u \in L$ одна единица потока может входить не более чем по одному ребру и выходить не более чем по одному ребру. Таким образом, одна единица положительного потока входит в u тогда и только тогда, когда существует в точности одна вершина $v \in R$, такая, что $f(u, v) = 1$, и из каждой вершины $u \in L$ выходит не более одного ребра, несущего положительный поток. Симметричные рассуждения применимы для каждой вершины $v \in R$. Следовательно, M является паросочетанием.

Чтобы показать, что $|M| = |f|$, заметим, что $f(s, u) = 1$ для каждой связанной вершины $u \in L$, и для каждого ребра $(u, v) \in E - M$ мы имеем $f(u, v) = 0$. Следовательно, $f(L \cup \{s\}, R \cup \{t\})$, чистый поток через разрез $(L \cup \{s\}, R \cup \{t\})$, равен $|M|$. Применив лемму 26.4, получаем, что $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ■

На основании леммы 26.9 можно сделать вывод о том, что максимальное паросочетание в двудольном графе G соответствует максимальному потоку в соответствующей ему транспортной сети G', следовательно, можно находить максимальное паросочетание в G с помощью алгоритма поиска максимального потока в G'. Единственной проблемой в данных рассуждениях является то, что алгоритм поиска максимального потока может вернуть такой поток в G', в котором некоторое значение $f(u, v)$ оказывается нецелым, несмотря на то что величина $|f|$ должна быть целой. В следующей теореме показано, что при использовании метода Форда–Фалкерсона такая проблема возникнуть не может.

Теорема 26.10 (Теорема о целочисленности)

Если функция пропускной способности c принимает только целые значения, то максимальный поток f, полученный с помощью метода Форда–Фалкерсона, обладает тем свойством, что значение потока $|f|$ является целочисленным. Более того, для всех вершин u и v величина $f(u, v)$ является целой.

Доказательство. Доказательство проводится индукцией по числу итераций. Его предлагается выполнить в качестве упр. 26.3.2. ■

Теперь мы можем доказать следствие из леммы 26.9.

Следствие 26.11

Мощность максимального паросочетания M в двудольном графе G равна величине максимального потока f в соответствующей транспортной сети G'.

Доказательство. Воспользуемся терминологией леммы 26.9. Предположим, что M представляет собой максимальное паросочетание в G, но соответствующий ему поток f в G' не максимален. Тогда в G' существует максимальный поток f', такой, что $|f'| > |f|$. Поскольку пропускные способности в G' являются целочисленными, теорема 26.10 позволяет считать поток f' целочисленным. Таким образом, f' соответствует некоторому паросочетанию M' в G мощностью $|M'| = |f'| > |f| = |M|$, что противоречит нашему предположению о том, что M является максимальным паросочетанием. Аналогично можно показать, что если f — максимальный поток в G', то соответствующее ему паросочетание является максимальным паросочетанием в G. ■

Таким образом, для заданного неориентированного двудольного графа G можно найти максимальное паросочетание путем создания транспортной сети G', применения метода Форда–Фалкерсона и непосредственного получения максимального паросочетания M по найденному максимальному целочисленному потоку f. Поскольку любое паросочетание в двудольном графе имеет мощность не более $\min(L, R) = O(V)$, величина максимального потока в G' составляет $O(V)$. Поэтому максимальное паросочетание в двудольном графе можно найти за время $O(VE') = O(VE)$, поскольку $|E'| = \Theta(E)$.

Упражнения

26.3.1

Примените алгоритм Форда–Фалкерсона для транспортной сети на рис. 26.8, (в) и покажите остаточную сеть после каждого увеличения потока. Вершины из множества L пронумеруйте сверху вниз от 1 до 5, а вершины множества R — от 6 до 9. Для каждой итерации укажите лексикографически наименьший увеличивающий путь.

26.3.2

Докажите теорему 26.10.

26.3.3

Пусть $G = (V, E)$ представляет собой двудольный граф с разбиением вершин $V = L \cup R$, а G' — соответствующая ему транспортная сеть. Найдите верхнюю границу длины любого увеличивающего пути, найденного в G' в процессе выполнения процедуры FORD-FULKERSON.

26.3.4 ★

Идеальное паросочетание (perfect matching) представляет собой паросочетание, в котором каждая вершина является связанной. Пусть $G = (V, E)$ является неориентированным двудольным графом с разбиением вершин $V = L \cup R$, где $|L| = |R|$. Для любого $X \subseteq V$ определим ***окрестность*** (neighborhood) X как

$$N(X) = \{y \in V : (x, y) \in E \text{ для некоторого } x \in X\} ,$$

т.е. это множество вершин, смежных с какой-либо из вершин X. Докажите ***теорему Холла*** (Hall's theorem): идеальное паросочетание в G существует тогда и только тогда, когда $|A| \leq |N(A)|$ для каждого подмножества $A \subseteq L$.

26.3.5 ★

Двудольный граф $G = (V, E)$, где $V = L \cup R$, называется ***d-регулярным*** (*d*-regular), если каждая вершина $v \in V$ имеет степень, в точности равную d. В каждом d-регулярном двудольном графе выполняется соотношение $|L| = |R|$. Докажите, что в каждом d-регулярном двудольном графе имеется паросочетание мощности $|L|$, показав, что минимальный разрез соответствующей транспортной сети имеет пропускную способность $|L|$.

★ 26.4. Алгоритмы проталкивания предпотока

В данном разделе будет рассмотрен подход к вычислению максимальных потоков, основанный на "проталкивании предпотока". В настоящее время многие асимптотически наиболее быстрые алгоритмы поиска максимального потока принадлежат данному классу, и на этом методе основаны реальные реализации алгоритмов поиска максимального потока. С помощью методов проталкивания предпотока можно решать и другие связанные с потоками задачи, например задачу поиска потока с минимальной стоимостью. В данном разделе приводится разработанный Голдбергом (Goldberg) "обобщенный" алгоритм поиска максимального потока, для которого существует простая реализация с временем выполнения $O(V^2E)$, что лучше времени работы алгоритма Эдмондса–Карпа $O(VE^2)$. В разделе 26.5 данный обобщенный алгоритм будет усовершенствован, что позволит получить алгоритм проталкивания предпотока, время выполнения которого составляет $O(V^3)$.

Алгоритмы проталкивания предпотока работают способом, более локальным, чем метод Форда–Фалкерсона. Вместо того чтобы для поиска увеличивающего пути анализировать всю остаточную сеть, алгоритмы проталкивания предпотока обрабатывают вершины по одной, рассматривая только соседей данной вершины в остаточной сети. Кроме того, в отличие от метода Форда–Фалкерсона, алгоритмы проталкивания предпотока не обеспечивают поддержание в процессе работы свойства сохранения потока. При этом, однако, они поддерживают ***предпоток*** (preflow), который представляет собой функцию $f : V \times V \to \mathbb{R}$, удовлетво-

ряющую ограничениям пропускной способности и следующему ослабленному условию сохранения потока:

$$\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \geq 0$$

для всех вершин $u \in V - \{s\}$. Будем называть величину

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \qquad (26.14)$$

избыточным потоком (excess flow), входящим в вершину u. Избыток в вершине представляет собой величину, на которую входящий поток превышает исходящий. Мы говорим, что вершина $u \in V - \{s,t\}$ ***переполненная*** (overflowing), если $e(u) > 0$.

Мы начнем данный раздел с описания интуитивных соображений, приводящих к методу проталкивания предпотока. Затем рассмотрим две применяемые в данном методе операции: "проталкивание" предпотока и подъем (перемаркировка — relabeling) некоторой вершины. Наконец, мы представим обобщенный алгоритм проталкивания предпотока и проанализируем его корректность и время выполнения.

Интуитивные соображения

Интуитивные соображения, лежащие в основе метода проталкивания предпотока, лучше всего проиллюстрировать на примере потоков жидкости: пусть транспортная сеть $G = (V,E)$ представляет собой систему труб с заданными пропускными способностями. Применив данную аналогию, о методе Форда–Фалкерсона можно сказать, что каждый увеличивающий путь в сети вызывает дополнительный поток жидкости без точек ветвления, который течет от истока к стоку. Метод Форда–Фалкерсона многократно добавляет дополнительные потоки до тех пор, пока дальнейшее добавление станет невозможным.

В основе обобщенного алгоритма проталкивания предпотока лежат другие интуитивные соображения. Пусть, как и ранее, ориентированные ребра соответствуют трубам. Вершины, в которых трубы пересекаются, обладают двумя интересными свойствами. Во-первых, чтобы принять избыточный поток, каждая вершина снабжена выходящей трубой, ведущей в произвольно большой резервуар, способный накапливать жидкость. Во-вторых, каждая вершина, ее резервуар и все трубные соединения находятся на платформе, высота которой увеличивается по мере работы алгоритма.

Высота вершины определяет, как проталкивается поток: поток может проталкиваться только вниз, т.е. от более высокой вершины к более низкой. Поток может быть направлен и от нижестоящей вершины к вышестоящей, но операции проталкивания потока проталкивают его только вниз. Высота истока является фиксированной и составляет $|V|$, а фиксированная высота стока равна 0. Высота всех других вершин сначала равна 0 и увеличивается со временем. Алгоритм сначала

посылает максимально возможный поток вниз от истока к стоку. Посылается количество, в точности достаточное для заполнения всех выходящих из истока труб до достижения их пропускной способности; таким образом посылается поток, равный пропускной способности разреза $(s, V - \{s\})$. Когда поток впервые входит в некоторую транзитную вершину, он накапливается в ее резервуаре. Отсюда он со временем проталкивается вниз.

Может случиться так, что все трубы, выходящие из вершины u и еще не заполненные потоком, ведут к вершинам, которые лежат на одном уровне с u или находятся выше нее. В этом случае, чтобы избавить переполненную вершину u от избыточного потока, необходимо увеличить ее высоту — провести операцию подъема (relabeling) вершины u. Ее высота увеличивается и становится на единицу больше, чем высота самой низкой из смежных с ней вершин, к которым ведут незаполненные трубы. Следовательно, после подъема вершины существует по крайней мере одна выходящая труба, по которой можно протолкнуть дополнительный поток.

В конечном итоге весь поток, который может пройти к стоку, оказывается там. Больше пройти не может, поскольку трубы подчиняются ограничениям пропускной способности; количество потока через любой разрез ограничено его пропускной способностью. Чтобы сделать предпоток "нормальным" потоком, алгоритм после этого посылает избытки, содержащиеся в резервуарах переполненных вершин, обратно к истоку, продолжая менять метки вершин, чтобы их высота превышала фиксированную высоту истока $|V|$. Как будет показано, после того как резервуары окажутся пустыми, предпоток станет не только нормальным, но и максимальным потоком.

Основные операции

Как следует из предыдущих рассуждений, в алгоритме проталкивания предпотока выполняются две основные операции: проталкивание избытка потока от вершины к одной из соседних с ней вершин и подъем вершины. Применение этих операций зависит от высот вершин, которым мы сейчас дадим более точные определения.

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t, а f является некоторым предпотоком в G. Функция $h : V \to \mathbb{N}$ является ***функцией высоты*** (height function)[3], $h(s) = |V|$, $h(t) = 0$ и

$$h(u) \leq h(v) + 1$$

для каждого остаточного ребра $(u, v) \in E_f$. Сразу же можно сформулировать следующую лемму.

---

[3]В литературе функция высоты обычно называется функцией расстояния (distance function), а высота вершины называется меткой расстояния (distance label). Мы используем термин "высота", поскольку он лучше согласуется с интуитивным обоснованием алгоритма. Высота вершины связана с ее расстоянием от стока t, которое можно найти с помощью поиска в ширину в G^{T}.

Лемма 26.12

Пусть $G = (V, E)$ представляет собой транспортную сеть, а f является некоторым предпотоком в G, и пусть h — функция высоты, заданная на множестве V. Для любых двух вершин $u, v \in V$ справедливо следующее утверждение: если $h(u) > h(v) + 1$, то (u, v) не является ребром остаточной сети. ■

Операция проталкивания

Основная операция PUSH(u, v) может применяться тогда, когда u является переполненной вершиной, $c_f(u, v) > 0$ и $h(u) = h(v) + 1$. Представленный ниже псевдокод обновляет предпоток f и избыточный поток для u и v. Предполагается, что остаточные пропускные способности $c_f(u, v)$ при заданных f и c можно вычислить за фиксированное время. Излишний поток в вершине u поддерживается в виде атрибута $u.e$, а высота вершины u — в виде атрибута $u.h$. Выражение $\Delta_f(u, v)$ представляет собой временную переменную, в которой хранится количество потока, которое можно протолкнуть из u в v.

PUSH(u, v)

1 // **Применяется при**: вершина u переполнена,
// $c_f(u, v) > 0$ и $u.h = v.h + 1$.
2 // **Действие:** проталкивает $\Delta_f(u, v) = \min(u.e, c_f(u, v))$
// единиц потока из u в v.
3 $\Delta_f(u, v) = \min(u.e, c_f(u, v))$
4 **if** $(u, v) \in E$
5 $\quad (u, v).f = (u, v).f + \Delta_f(u, v)$
6 **else** $(v, u).f = (v, u).f - \Delta_f(u, v)$
7 $u.e = u.e - \Delta_f(u, v)$
8 $v.e = v.e + \Delta_f(u, v)$

Процедура PUSH работает следующим образом. Поскольку вершина u имеет положительный избыток $u.e$ и остаточная пропускная способность ребра (u, v) положительна, можно увеличить поток из u в v на величину $\Delta_f(u, v) = \min(u.e, c_f(u, v))$, при этом избыток $u.e$ не становится отрицательным и не будет превышена пропускная способность $c(u, v)$. В строке 3 вычисляется значение $\Delta_f(u, v)$, после чего в строках 4–6 обновляется f. В строке 5 увеличивается поток в ребре (u, v), поскольку мы проталкиваем поток через остаточное ребро, которое также является и исходным ребром. В строке 6 уменьшается поток в ребре (v, u), поскольку остаточное ребро в действительности обратно ребру в исходной сети. Наконец в строках 7 и 8 обновляются избыточные потоки в вершины u и v. Таким образом, если f являлся предпотоком перед вызовом процедуры PUSH, он останется предпотоком и после ее выполнения.

Обратите внимание, что в коде процедуры PUSH ничто не зависит от высот вершин u и v; тем не менее мы запретили вызов процедуры, если не выполнено условие $u.h = v.h + 1$. Таким образом, избыточный поток проталкивается вниз только при разности высот, равной 1. Согласно лемме 26.12 между двумя вершинами, высоты которых отличаются более чем на 1, не существует остаточных

ребер, а значит, поскольку атрибут h является функцией высоты, мы ничего не добьемся, разрешив проталкивать вниз поток при разности высот, превышающей 1.

Процедура PUSH(u, v) называется ***проталкиванием*** (push) из u в v. Если операция проталкивания применяется к некоторому ребру (u, v), выходящему из вершины u, будем говорить, что операция проталкивания применяется к u. Если в результате ребро (u, v) становится ***насыщенным*** (saturated) (после проталкивания $c_f(u, v) = 0$), то это ***насыщающее проталкивание*** (saturating push), в противном случае это ***ненасыщающее проталкивание*** (nonsaturating push). Если ребро становится насыщенным, оно исчезает из остаточной сети. Один из результатов ненасыщающего проталкивания характеризует следующая лемма.

Лемма 26.13

После ненасыщающего проталкивания из u в v вершина u более не является переполненной.

Доказательство. Поскольку проталкивание ненасыщающее, фактическое количество посланного потока $\Delta_f(u, v)$ должно быть равно величине $u.e$ непосредственно перед проталкиванием. Поскольку избыток $u.e$ уменьшается на эту величину, после проталкивания он становится равным 0. ■

Операция подъема

Основная операция RELABEL(u) применяется, если вершина u переполнена и если $u.h \le v.h$ для всех ребер $(u, v) \in E_f$. Иными словами, переполненную вершину u можно подвергнуть подъему, если все вершины v, для которых имеется остаточная пропускная способность от u к v, расположены не ниже u, так что протолкнуть поток из u нельзя. (Напомним, что по определению ни исток s, ни сток t не могут быть переполнены; следовательно, ни s, ни t нельзя подвергать подъему.)

RELABEL(u)

// Применяется при: вершина u переполнена, и для всех $v \in V$, таких,
// что $(u, v) \in E_f$, имеем $u.h \le v.h$.
// Действие: увеличивает высоту u.
$u.h = 1 + \min\{v.h : (u, v) \in E_f\}$

Когда вызывается операция RELABEL(u), мы говорим, что вершина u подвергается ***подъему*** (relabeled). Заметим, что когда выполняется подъем u, E_f должно содержать хотя бы одно ребро, выходящее из u, чтобы минимизация в коде операции осуществлялась по непустому множеству. Это свойство вытекает из предположения о том, что вершина u переполнена, что, в свою очередь, говорит нам, что

$$u.e = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) > 0 .$$

Поскольку все потоки неотрицательны, должна быть по крайней мере одна вершина v, такая, что $(v, u).f > 0$. Но тогда $c_f(u, v) > 0$, откуда вытекает, что

$(u, v) \in E_f$. Таким образом, операция RELABEL(u) назначает u наибольшую высоту, допускаемую наложенными на функцию высоты ограничениями.

Обобщенный алгоритм

Обобщенный алгоритм проталкивания предпотока использует следующую подпрограмму для создания начального предпотока в транспортной сети.

```
INITIALIZE-PREFLOW(G, s)
for каждой вершины v ∈ G.V
    v.h = 0
    v.e = 0
for каждого ребра (u, v) ∈ G.E
    (u, v).f = 0
s.h = |G.V|
for каждой вершины v ∈ s.Adj
    (s, v).f = c(s, v)
    v.e = c(s, v)
    s.e = s.e − c(s, v)
```

INITIALIZE-PREFLOW создает начальный предпоток f, определяемый как

$$(u, v).f = \begin{cases} c(u, v) , & \text{если } u = s , \\ 0 & \text{в противном случае} . \end{cases} \qquad (26.15)$$

Иначе говоря, каждое ребро, выходящее из истока s, заполняется до его пропускной способности, а все остальные ребра не несут потока. Для каждой вершины v, смежной с истоком, изначально мы имеем $v.e = c(s, v)$ и инициализируем $s.e$ суммой этих значений с обратным знаком. Обобщенный алгоритм начинает работу с начальной функцией высоты h, задаваемой следующим образом:

$$u.h = \begin{cases} |V| , & \text{если } u = s , \\ 0 & \text{в противном случае} . \end{cases} \qquad (26.16)$$

Уравнение (26.16) определяет функцию высоты, поскольку единственными ребрами (u, v), для которых $u.h > v.h + 1$, являются те, для которых $u = s$, и эти ребра заполнены, а это означает, что их нет в остаточной сети.

Инициализация, за которой следует ряд операций проталкивания и подъема, выполняемых без определенного порядка, образует алгоритм GENERIC-PUSH-RELABEL.

```
GENERIC-PUSH-RELABEL(G)
INITIALIZE-PREFLOW(G, s)
while существует применимая операция
            проталкивания или подъема
    выбрать и выполнить операцию проталкивания или подъема
```

В следующей лемме утверждается, что до тех пор, пока существует хотя бы одна переполненная вершина, применима хотя бы одна из этих операций.

Лемма 26.14 (Для переполненной вершины можно выполнить либо проталкивание, либо подъем)

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t, а f является предпотоком, и пусть h является произвольной функцией веса для f. Если u представляет собой произвольную переполненную вершину, то к ней применимо либо проталкивание, либо подъем.

Доказательство. Для любого остаточного ребра (u, v) выполняется соотношение $h(u) \leq h(v) + 1$, поскольку h представляет собой функцию высоты. Если к переполненной вершине u не применима операция проталкивания, то для всех остаточных ребер (u, v) должно выполняться условие $h(u) < h(v) + 1$, откуда следует, что $h(u) \leq h(v)$. В таком случае к u можно применить операцию подъема. ■

Корректность метода проталкивания предпотока

Чтобы показать, что обобщенный алгоритм проталкивания предпотока позволяет решить задачу максимального потока, сначала докажем, что если он завершается, то предпоток f является максимальным потоком. Затем докажем, что алгоритм завершается. Начнем с рассмотрения некоторых свойств функции высоты h.

Лемма 26.15 (Высота вершины никогда не уменьшается)

При выполнении процедуры GENERIC-PUSH-RELABEL над транспортной сетью $G = (V, E)$ для любой вершины $u \in V$ ее высота $u.h$ никогда не уменьшается. Более того, всякий раз, когда к вершине u применяется операция подъема, ее высота $u.h$ увеличивается как минимум на 1.

Доказательство. Поскольку высота вершины меняется только при выполнении операции подъема, достаточно доказать второе утверждение леммы. Если вершина u должна подвергнуться подъему, то для всех вершин v, таких, что $(u, v) \in E_f$, выполняется условие $u.h \leq v.h$. Таким образом, $u.h < 1 + \min\{v.h : (u, v) \in E_f\}$, и операция должна увеличить значение $u.h$. ■

Лемма 26.16

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t. Во время выполнения процедуры GENERIC-PUSH-RELABEL над сетью G атрибут h сохраняет свойства функции высоты.

Доказательство. Доказательство проводится индукцией по числу выполненных основных операций. Изначально, как уже отмечалось, h является функцией высоты.

Мы утверждаем, что если h является функцией высоты, то операция RELABEL(u) оставляет h функцией высоты. Если рассмотреть остаточное ребро $(u, v) \in E_f$, которое выходит из u, то операция RELABEL(u) гарантирует,

что после нее будет выполняться соотношение $u.h \leq v.h + 1$. Теперь рассмотрим остаточное ребро (w, u), входящее в u. Согласно лемме 26.15 из $w.h \leq u.h + 1$ перед выполнением операции RELABEL(u) вытекает $w.h < u.h + 1$ после нее. Таким образом, операция RELABEL(u) оставляет h функцией высоты.

Теперь рассмотрим операцию PUSH(u, v). Эта операция может добавить ребро (v, u) к E_f, и может удалить ребро (u, v) из E_f. В первом случае имеем $v.h = u.h - 1 < u.h + 1$, так что h остается функцией высоты. Во втором случае удаление ребра (u, v) из остаточной сети приводит к удалению соответствующего ограничения, так что h по-прежнему остается функцией высоты. ■

Следующая лемма характеризует важное свойство функций высоты.

Лемма 26.17

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t, f является предпотоком в G, а h — функцией высоты, определенной на множестве V. Тогда в остаточной сети G_f не существует пути из истока s в сток t.

Доказательство. Предположим, что в G_f имеется некоторый путь $p = \langle v_0, v_1, \ldots, v_k \rangle$ из s в t, где $v_0 = s$, а $v_k = t$, и покажем, что это приводит к противоречию. Без потери общности можно считать, что p — простой путь, так что $k < |V|$. Для $i = 0, 1, \ldots, k - 1$ ребра $(v_i, v_{i+1}) \in E_f$. Поскольку h является функцией высоты, для $i = 0, 1, \ldots, k - 1$ справедливы соотношения $h(v_i) \leq h(v_{i+1}) + 1$. Объединив эти неравенства вдоль пути p, получаем, что $h(s) \leq h(t) + k$. Но поскольку $h(t) = 0$, получаем $h(s) \leq k < |V|$, что противоречит требованию $h(s) = |V|$ к функции высоты. ■

Теперь мы готовы показать, что после завершения обобщенного алгоритма проталкивания предпотока вычисленный алгоритмом предпоток является максимальным потоком.

Теорема 26.18 (Корректность обобщенного алгоритма проталкивания предпотока)

Если алгоритм GENERIC-PUSH-RELABEL, выполняемый над транспортной сетью $G = (V, E)$ с истоком s и стоком t, завершается, то вычисленный им предпоток f является максимальным потоком в G.

Доказательство. Воспользуемся следующим инвариантом цикла.

> Всякий раз, когда в строке 2 процедуры GENERIC-PUSH-RELABEL выполняется проверка условия цикла **while**, f является предпотоком.

Инициализация. INITIALIZE-PREFLOW делает f предпотоком.

Сохранение. Внутри цикла **while** в строках 2 и 3 выполняются только операции проталкивания и подъема. Операции подъема влияют только на атрибуты высоты, но не на величины потока, следовательно, от них не зависит, будет ли f предпотоком. Анализируя работу процедуры PUSH, мы доказали (с. 778), что

если f является предпотоком перед выполнением операции проталкивания, он остается предпотоком и после ее выполнения.

Завершение. По завершении процедуры каждая вершина из множества $V - \{s, t\}$ должна иметь нулевой избыток, поскольку из леммы 26.14 и инварианта, что f всегда остается предпотоком, вытекает, что переполненных вершин нет. Следовательно, f является потоком. Лемма 26.16 показывает, что при завершении h является функцией высоты; таким образом, согласно лемме 26.17 в остаточной сети G_f не существует пути из s в t. Согласно теореме о максимальном потоке и минимальном разрезе (теорема 26.6) f является максимальным потоком. ■

Анализ метода проталкивания предпотока

Чтобы показать, что обобщенный алгоритм проталкивания предпотока действительно завершается, найдем границу количества выполняемых им операций. Для каждого вида операций (подъем, насыщающее проталкивание и ненасыщающее проталкивание) имеется своя граница. Зная эти границы, несложно построить алгоритм, время работы которого — $O(V^2E)$. Однако, прежде чем проводить анализ, докажем важную лемму. Вспомним, что мы разрешаем ребрам входить в исток в остаточной сети.

Лемма 26.19

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t, а f является предпотоком в G. Тогда в остаточной сети G_f для любой переполненной вершины x существует простой путь из x в s.

Доказательство. Для переполненной вершины x введем $U = \{v :$ существует простой путь из x в v в $G_f\}$. Теперь предположим, что $s \notin U$, и покажем, что это приводит к противоречию. Обозначим $\overline{U} = V - U$.

Возьмем определение избытка из уравнения (26.14), выполним суммирование по всем вершинам в U и заметим, что $V = U \cup \overline{U}$. Это даст

$$\begin{aligned}
&\sum_{u \in U} e(u) \\
&\quad = \sum_{u \in U} \left(\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \right) \\
&\quad = \sum_{u \in U} \left(\left(\sum_{v \in U} f(v,u) + \sum_{v \in \overline{U}} f(v,u) \right) - \left(\sum_{v \in U} f(u,v) + \sum_{v \in \overline{U}} f(u,v) \right) \right) \\
&\quad = \sum_{u \in U} \sum_{v \in U} f(v,u) + \sum_{u \in U} \sum_{v \in \overline{U}} f(v,u) - \sum_{u \in U} \sum_{v \in U} f(u,v) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u,v) \\
&\quad = \sum_{u \in U} \sum_{v \in \overline{U}} f(v,u) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u,v) .
\end{aligned}$$

Мы знаем, что величина $\sum_{u \in U} e(u)$ должна быть положительной, поскольку $e(x) > 0$, $x \in U$, что все вершины, отличные от s, имеют неотрицательный избыток и что согласно нашему предположению $s \notin U$. Таким образом, имеем

$$\sum_{u \in U} \sum_{v \in \overline{U}} f(v,u) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u,v) > 0 . \qquad (26.17)$$

Все потоки ребер неотрицательны, так что, чтобы выполнялось уравнение (26.17), необходимо иметь $\sum_{u \in U} \sum_{v \in \overline{U}} f(v,u) > 0$. Следовательно, должна существовать как минимум одна пара вершин $u' \in U$ и $v' \in \overline{U}$, обладающих тем свойством, что $f(v',u') > 0$. Но если $f(v',u') > 0$, должно существовать остаточное ребро (u',v'), а это означает, что имеется простой путь из x в v' (путь $x \rightsquigarrow u' \to v'$), что противоречит определению U. ■

В следующей лемме устанавливаются границы высот вершин, а в вытекающем из нее следствии устанавливается предел общего числа выполненных операций подъема.

Лемма 26.20

Пусть $G = (V,E)$ представляет собой транспортную сеть с истоком s и стоком t. В любой момент в процессе выполнения процедуры GENERIC-PUSH-RELABEL над сетью G для всех вершин $u \in V$ выполняется соотношение $u.h \leq 2|V| - 1$.

Доказательство. Высоты истока s и стока t никогда не изменяются, поскольку эти вершины по определению не переполняются. Таким образом, всегда $s.h = |V|$ и $t.h = 0$, причем оба значения не превышают $2|V| - 1$.

Рассмотрим теперь произвольную вершину $u \in V - \{s,t\}$. Изначально $u.h = 0 \leq 2|V| - 1$. Покажем, что после каждого подъема неравенство $u.h \leq 2|V| - 1$ остается справедливым. При подъеме вершины u она является переполненной, и согласно лемме 26.19 имеется простой путь p из u в s в G_f. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = u$, $v_k = s$ и $k \leq |V| - 1$, поскольку p — простой путь. Для $i = 0, 1, \ldots, k-1$ имеем $(v_i, v_{i+1}) \in E_f$, а следовательно, $v_i.h \leq v_{i+1}.h + 1$ согласно лемме 26.16. Расписав неравенства для всех составляющих пути p, получаем $u.h = v_0.h \leq v_k.h + k \leq s.h + (|V| - 1) = 2|V| - 1$. ■

Следствие 26.21 (Верхний предел числа подъемов)

Пусть $G = (V,E)$ представляет собой транспортную сеть с истоком s и стоком t. Тогда в процессе выполнения процедуры GENERIC-PUSH-RELABEL над G число подъемов не превышает $2|V| - 1$ на вершину, а их общее количество — не более $(2|V| - 1)(|V| - 2) < 2|V|^2$.

Доказательство. В множестве $V - \{s,t\}$ могут быть подняты только $|V| - 2$ вершин. Пусть $u \in V - \{s,t\}$. Операция RELABEL(u) увеличивает высоту $u.h$. Значение $u.h$ изначально равно 0 и согласно лемме 26.20 возрастает не более чем до $2|V| - 1$. Таким образом, каждая вершина $u \in V - \{s,t\}$ подвергается

подъему не более $2|V| - 1$ раз, а общее число выполненных подъемов не превышает $(2|V| - 1)(|V| - 2) < 2|V|^2$. ■

Лемма 26.20 помогает также определить границу количества насыщающих проталкиваний.

Лемма 26.22 (Граница количества насыщающих проталкиваний)

В процессе выполнения алгоритма GENERIC-PUSH-RELABEL над любой транспортной сетью $G = (V, E)$ число насыщающих проталкиваний меньше, чем $2|V||E|$.

Доказательство. Для любой пары вершин $u, v \in V$ рассмотрим насыщающие проталкивания от u к v и от v к u (в обе стороны) и назовем их насыщающими проталкиваниями между u и v. Если есть хотя бы одно такое проталкивание, то хотя бы одно из ребер (u, v) и (v, u) является ребром из E. Теперь предположим, что произошло насыщающее проталкивание из u в v. В этот момент $v.h = u.h - 1$. Чтобы позднее могло произойти еще одно проталкивание из u в v, алгоритм сначала должен протолкнуть поток из v в u, что невозможно до тех пор, пока не будет выполнено условие $v.h = u.h + 1$. Поскольку $u.h$ никогда не уменьшается, для того чтобы выполнялось условие $v.h = u.h + 1$, значение $v.h$ должно увеличиться по меньшей мере на 2. Аналогично $u.h$ должно увеличиться между последовательными насыщающими проталкиваниями из v в u как минимум на 2. Высота изначально принимает значение 0 и согласно лемме 26.20 никогда не превышает $2|V| - 1$, откуда следует, что количество раз, когда высота вершины может увеличиться на 2, меньше $|V|$. Поскольку между двумя насыщающими проталкиваниями между u и v хотя бы одна из высот $u.h$ и $v.h$ должна увеличиться на 2, всего имеется меньше $2|V|$ насыщающих проталкиваний между u и v.

Умножив это число на число ребер, получим, что общее число насыщающих проталкиваний меньше, чем $2|V||E|$. ■

Очередная лемма устанавливает границу числа ненасыщающих проталкиваний в обобщенном алгоритме проталкивания предпотока.

Лемма 26.23 (Граница количества ненасыщающих проталкиваний)

В процессе выполнения алгоритма GENERIC-PUSH-RELABEL над любой транспортной сетью $G = (V, E)$ число ненасыщающих проталкиваний меньше $4|V|^2(|V| + |E|)$.

Доказательство. Определим функцию потенциала следующим образом: $\Phi = \sum_{v:e(v)>0} v.h$. Изначально $\Phi = 0$, и значение Φ может изменяться после каждого подъема, насыщающего и ненасыщающего проталкивания. Найдем предел величины, на которую насыщающие проталкивания и подъемы могут увеличивать Φ. Затем покажем, что каждое ненасыщающее проталкивание должно уменьшать Φ как минимум на 1 и используем эти оценки для определения верхней границы числа ненасыщающих проталкиваний.

Рассмотрим два пути увеличения Φ. Во-первых, подъем вершины u увеличивает Φ менее чем на $2|V|$, поскольку множество, для которого вычисляется сумма, остается прежним, а подъем не может увеличить высоту вершины u больше, чем ее максимально возможная высота, которая составляет не более $2|V|-1$ согласно лемме 26.20. Во-вторых, насыщающее проталкивание из вершины u в вершину v увеличивает Φ менее чем на $2|V|$, поскольку никаких изменений высот при этом не происходит, и только вершина v, высота которой не более $2|V|-1$, может стать переполненной.

Теперь покажем, что ненасыщающее проталкивание из u в v уменьшает Φ не менее чем на 1. Почему? Перед ненасыщающим проталкиванием вершина u была переполненной, а v могла быть переполненной или непереполненной. Согласно лемме 26.13 после этого проталкивания u больше не является переполненной. Кроме того, если только v не является истоком, она может быть как переполненной, так и не быть таковой после проталкивания. Следовательно, потенциальная функция Φ уменьшилась ровно на $u.h$, а увеличилась на 0 или на $v.h$. Поскольку $u.h - v.h = 1$, в итоге потенциальная функция уменьшается как минимум на 1.

Итак, в ходе выполнения алгоритма увеличение Φ происходит благодаря подъемам и насыщающим проталкиваниям; согласно следствию 26.21 и лемме 26.22 это увеличение ограничено и составляет менее $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V|+|E|)$. Поскольку $\Phi \geq 0$, суммарная величина уменьшения и, следовательно, общее число ненасыщающих проталкиваний меньше, чем $4|V|^2(|V|+|E|)$. ∎

Определив границу числа подъемов, насыщающего проталкивания и ненасыщающего проталкивания, мы заложили основу дальнейшего анализа процедуры GENERIC-PUSH-RELABEL, а следовательно, любых других алгоритмов, основанных на методе проталкивания предпотока.

Теорема 26.24

При выполнении процедуры GENERIC-PUSH-RELABEL над любой транспортной сетью $G = (V, E)$ число основных операций составляет $O(V^2E)$.

Доказательство. Непосредственно вытекает из следствия 26.21 и лемм 26.22 и 26.23. ∎

Таким образом, алгоритм завершается после $O(V^2E)$ операций. Итак, осталось предложить эффективные методы реализации каждой операции и выбора подходящей выполняемой операции.

Следствие 26.25

Существует реализация обобщенного алгоритма проталкивания предпотока, которая для любой транспортной сети $G = (V, E)$ выполняется за время $O(V^2E)$.

Доказательство. В упр. 26.4.2 предлагается показать, как реализовать обобщенный алгоритм, в котором на каждый подъем затрачивается время $O(V)$, а на каждое проталкивание — $O(1)$. Там же предлагается разработать структуру дан-

ных, которая позволит выбирать применимую операцию за время $O(1)$. Тем самым следствие будет доказано. ■

Упражнения

26.4.1

Докажите, что после завершения процедуры INITIALIZE-PREFLOW(G, s) мы имеем $s.e \leq -|f^*|$, где f^* представляет собой максимальный поток в G.

26.4.2

Покажите, как реализовать обобщенный алгоритм проталкивания предпотока, в котором на каждый подъем затрачивается время $O(V)$, на каждое проталкивание — $O(1)$, и то же время $O(1)$ требуется для выбора применимой операции; суммарное время выполнения при этом составляет $O(V^2E)$.

26.4.3

Докажите, что время, затрачиваемое в целом на выполнение всех $O(V^2)$ подъемов в обобщенном алгоритме проталкивания предпотока, составляет только $O(VE)$.

26.4.4

Предположим, что с помощью алгоритма проталкивания предпотока найден максимальный поток для транспортной сети $G = (V, E)$. Разработайте быстрый алгоритм поиска минимального разреза в G.

26.4.5

Разработайте эффективный алгоритм проталкивания предпотока для поиска максимального паросочетания в двудольном графе. Проанализируйте время его работы.

26.4.6

Предположим, что все пропускные способности ребер транспортной сети $G = (V, E)$ принадлежат множеству $\{1, 2, \ldots, k\}$. Проанализируйте время выполнения обобщенного алгоритма проталкивания предпотока, выразив его через $|V|$, $|E|$ и k. (*Указание:* сколько ненасыщающих проталкиваний можно применить к каждому ребру, прежде чем оно станет насыщенным?)

26.4.7

Покажите, что можно заменить строку 6 процедуры INITIALIZE-PREFLOW строкой

6 $s.h = |G.V| - 2$

без влияния на корректность или асимптотическую производительность обобщенного алгоритма проталкивания предпотока.

26.4.8

Пусть $\delta_f(u, v)$ представляет собой расстояние (количество ребер) от u до v в остаточной сети G_f. Покажите, что в процессе работы процедуры GENERIC-PUSH-

RELABEL выполняются следующие свойства: из $u.h < |V|$ вытекает $u.h \leq \delta_f(u,t)$, а из $u.h \geq |V|$ следует $u.h - |V| \leq \delta_f(u,s)$.

26.4.9 ★

Пусть, как и в предыдущем упражнении, $\delta_f(u,v)$ представляет собой расстояние от u до v в остаточной сети G_f. Покажите, как можно модифицировать обобщенный алгоритм проталкивания предпотока, чтобы в процессе работы процедуры поддерживались следующие свойства: из $u.h < |V|$ вытекает $u.h = \delta_f(u,t)$, а из $u.h \geq |V|$ следует $u.h - |V| = \delta_f(u,s)$. Суммарное время, затраченное на обеспечение выполнения данного свойства, должно составлять $O(VE)$.

26.4.10

Покажите, что количество ненасыщающих проталкиваний, выполняемых процедурой GENERIC-PUSH-RELABEL в транспортной сети $G = (V,E)$ для $|V| \geq 4$ не превышает $4|V|^2|E|$.

★ 26.5. Алгоритм "поднять-в-начало"

Метод проталкивания предпотока позволяет применять основные операции в произвольном порядке. Однако путем тщательного выбора порядка их выполнения и при эффективном управлении структурой сетевых данных можно решить задачу поиска максимального потока быстрее, чем за предельное время $O(V^2E)$, определенное следствием 26.25. Далее мы рассмотрим алгоритм "поднять-в-начало" (relabel-to-front), основанный на методе проталкивания предпотока, время выполнения которого составляет $O(V^3)$, что асимптотически не хуже, чем $O(V^2E)$, а для плотных сетей даже лучше.

Алгоритм "поднять-в-начало" поддерживает список вершин сети. Алгоритм многократно сканирует список с самого начала, выбирает некоторую переполненную вершину u, а затем "разгружает" ее, т.е. выполняет операции проталкивания и подъема до тех пор, пока избыток в u не перестанет быть положительным. Если выполнялось поднятие вершины, то она переносится в начало списка (отсюда и название алгоритма: "поднять-в-начало"), и алгоритм начинает сканирование списка заново.

Для исследования корректности и временны́х характеристик данного алгоритма используется понятие "допустимых" ребер: это ребра остаточной сети, через которые можно протолкнуть поток. Доказав некоторые свойства сети, состоящей из допустимых ребер, мы рассмотрим операцию разгрузки, а затем представим и проанализируем сам алгоритм "поднять-в-начало".

Допустимые ребра и сети

Если $G = (V,E)$ представляет собой некоторую транспортную сеть с истоком s и стоком t, f — предпоток в G, а h — функция высоты, то мы говорим, что ребро (u,v) является ***допустимым ребром*** (admissible edge), если $c_f(u,v) > 0$

и $h(u) = h(v) + 1$. В противном случае ребро (u, v) называется ***недопустимым*** (inadmissible). ***Допустимой сетью*** (admissible network) является сеть $G_{f,h} = (V, E_{f,h})$, где $E_{f,h}$ — множество допустимых ребер.

Допустимая сеть состоит из тех ребер, через которые можно протолкнуть поток. Следующая лемма показывает, что такая сеть является ориентированным ациклическим графом.

Лемма 26.26 (Допустимая сеть является ациклической)

Если $G = (V, E)$ является транспортной сетью, f представляет собой предпоток в G, а h — функция высоты на G, то допустимая сеть $G_{f,h} = (V, E_{f,h})$ ациклична.

Доказательство. Проведем доказательство методом от противного. Предположим, что $G_{f,h}$ содержит некоторый цикл $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = v_k$ и $k > 0$. Поскольку каждое ребро в p является допустимым, справедливо равенство $h(v_{i-1}) = h(v_i) + 1$ для $i = 1, 2, \ldots, k$. Просуммировав эти равенства вдоль циклического пути, получаем

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i) + 1)$$
$$= \sum_{i=1}^{k} h(v_i) + k \; .$$

Поскольку каждая вершина цикла p встречается при суммировании по одному разу, приходим к выводу, что $0 = k$, что противоречит первоначальному предположению. ■

В двух следующих леммах показано, как операции проталкивания и подъема изменяют допустимую сеть.

Лемма 26.27

Пусть $G = (V, E)$ представляет собой транспортную сеть, f — предпоток в G, и предположим, что атрибут h является функцией высоты. Если вершина u переполнена, а (u, v) является допустимым ребром, то применима процедура PUSH(u, v). Эта операция не создает новых допустимых ребер, но может привести к тому, что ребро (u, v) станет недопустимым.

Доказательство. По определению допустимого ребра из u в v можно протолкнуть поток. Поскольку вершина u переполнена, применяется операция PUSH(u, v). В результате проталкивания потока из u в v может быть создано только одно новое остаточное ребро (v, u). Поскольку $v.h = u.h - 1$, ребро (v, u) не может стать допустимым. Если примененная операция является насыщающим проталкиванием, то после ее выполнения $c_f(u, v) = 0$ и ребро (u, v) становится недопустимым. ■

Лемма 26.28

Пусть $G = (V, E)$ представляет собой транспортную сеть, f является предпотоком в G, и предположим, что атрибут h является функцией высоты. Если вершина u переполнена и не имеется допустимых ребер, выходящих из u, то применяется операция RELABEL(u). После подъема появляется по крайней мере одно допустимое ребро, выходящее из u, но нет допустимых ребер, входящих в u.

Доказательство. Если вершина u переполнена, то согласно лемме 26.14 к ней применяется или операция проталкивания, или операция подъема. Если не существует допустимых ребер, выходящих из u, то протолкнуть поток из u невозможно, следовательно, применяется операция RELABEL(u). После данного подъема $u.h = 1 + \min\{v.h : (u, v) \in E_f\}$. Таким образом, если v — вершина, в которой реализуется минимум указанного множества, ребро (u, v) становится допустимым. Следовательно, после подъема имеется по крайней мере одно допустимое ребро, выходящее из u.

Чтобы показать, что после подъема не существует входящих в u допустимых ребер, предположим, что существует некоторая вершина v, такая, что ребро (v, u) допустимо. Тогда после подъема $v.h = u.h + 1$, так что непосредственно перед подъемом $v.h > u.h + 1$. Но согласно лемме 26.12 не существует остаточных ребер между вершинами, высоты которых отличаются более чем на 1. Кроме того, подъем вершины не меняет остаточную сеть. Таким образом, ребро (v, u) не принадлежит остаточной сети, а следовательно, оно не может находиться в допустимой сети. ■

Списки соседей

Ребра в алгоритме "поднять-в-начало" объединены в так называемые "списки соседей". В заданной транспортной сети $G = (V, E)$ ***списком соседей*** (neighbor list) $u.N$ некоторой вершины $u \in V$ является односвязный список вершин, смежных с u в G. Таким образом, вершина v оказывается в списке $u.N$, если $(u, v) \in E$ или $(v, u) \in E$. Список соседей $u.N$ содержит только такие вершины v, для которых может существовать остаточное ребро (u, v). На первую вершину в списке $u.N$ указывает указатель $u.N.head$. Указатель $v.next\text{-}neighbor$ указывает на вершину, следующую в списке соседей за v; этот указатель имеет значение NIL, если v является последней вершиной в списке соседей.

Алгоритм "поднять-в-начало" циклически обрабатывает каждый список соседей в произвольном порядке, который фиксируется в процессе выполнения алгоритма. Для каждой вершины u атрибут $u.current$ указывает на текущую вершину списка $u.N$. Изначально $u.current$ устанавливается равным $u.N.head$.

Разгрузка переполненной вершины

Переполненная вершина u ***разгружается*** (discharged) путем проталкивания всего ее избыточного потока через допустимые ребра в смежные вершины, при этом, если необходимо, выполняется подъем вершины u, чтобы ребра, выходящие

из вершины u, стали допустимыми. Псевдокод разгрузки выглядит следующим образом.

```
DISCHARGE(u)
1  while u.e > 0
2      v = u.current
3      if v == NIL
4          RELABEL(u)
5          u.current = u.N.head
6      elseif c_f(u, v) > 0 и u.h == v.h + 1
7          PUSH(u, v)
8      else u.current = v.next-neighbor
```

На рис. 26.9 показаны несколько итераций цикла **while** (строки 1–8), тело которого выполняется до тех пор, пока вершина u имеет положительный избыток. Каждая итерация выполняет одно из трех действий в зависимости от текущей вершины v из списка соседей $u.N$.

1. Если v равно NIL, значит, мы дошли до конца списка $u.N$. Выполняется подъем вершины u (строка 4), а затем (строка 5) текущей соседней вершиной u делается первая вершина из списка $u.N$. (В лемме 26.29 утверждается, что в данной ситуации подъем применим.)
2. Если v не равно NIL и ребро (u, v) является допустимым (что определяется с помощью проверки в строке 6), то (строка 7) выполняется проталкивание части (или всего) избытка из u в вершину v.
3. Если v не равно NIL, но ребро (u, v) является недопустимым, то (строка 8) указатель $u.current$ в списке $u.N$ перемещается на одну позицию вперед.

Заметим, что если процедура DISCHARGE вызывается для некоторой переполненной вершины u, последним действием, выполняемым данной процедурой, должно быть проталкивание из u. Почему? Процедура завершается только тогда, когда избыток $u.e$ становится равным нулю, и ни подъем, ни перемещение указателя $u.current$ не влияют на значение $u.e$.

Теперь необходимо убедиться, что когда процедура DISCHARGE вызывает процедуры PUSH или RELABEL, эти операции применимы. В следующей лемме доказывается данный факт.

Лемма 26.29

Если процедура DISCHARGE вызывает в строке 7 процедуру PUSH(u, v), то к (u, v) применима операция проталкивания. Если процедура DISCHARGE вызывает в строке 4 процедуру RELABEL(u), к вершине u применим подъем.

Доказательство. Проверки в строках 1 и 6 гарантируют, что операция проталкивания вызывается только тогда, когда она применима; таким образом, первое утверждение леммы доказано.

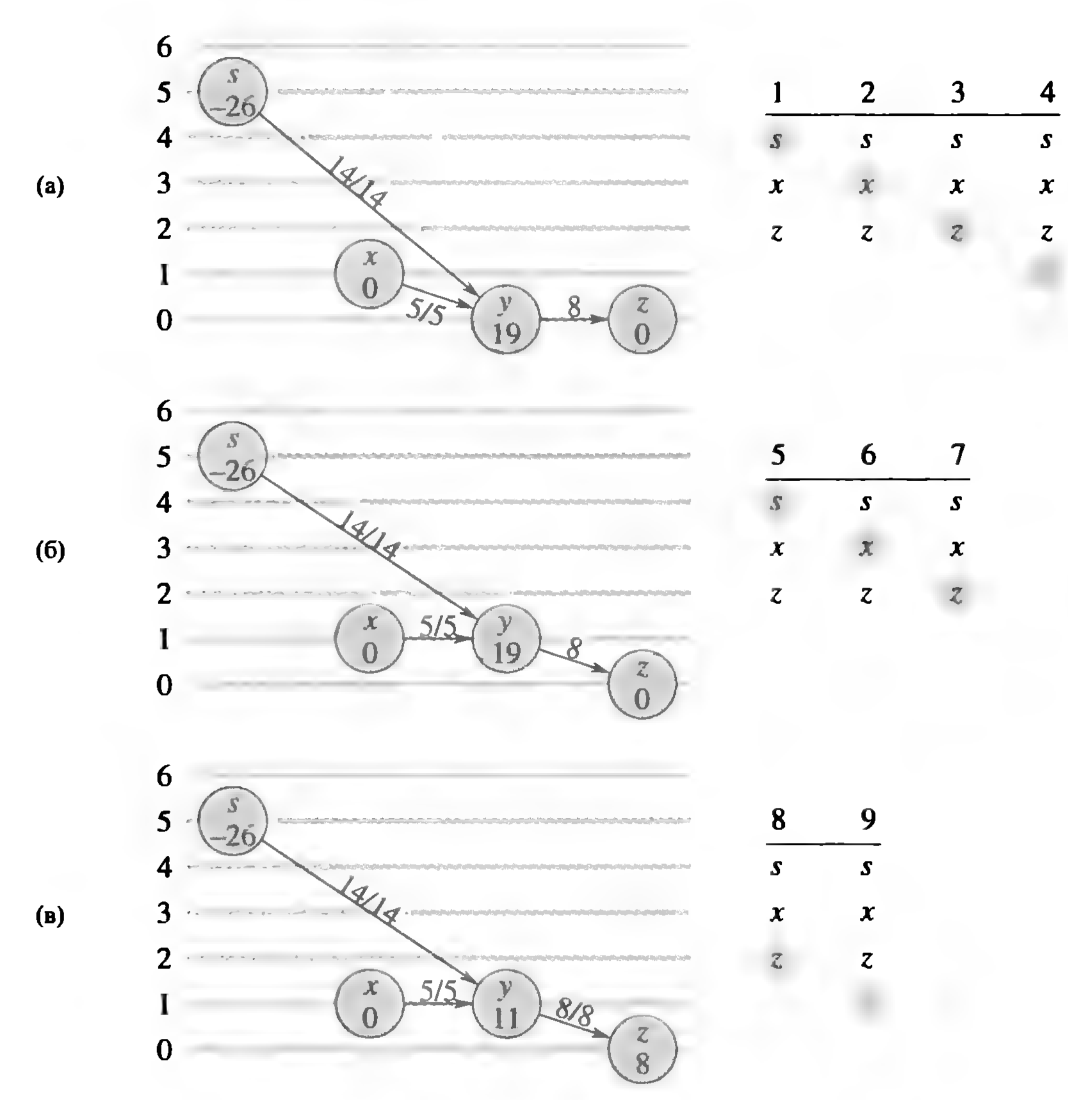


Рис. 26.9. Разгрузка вершины y. Она требует 15 итераций цикла **while** процедуры DISCHARGE для того, чтобы протолкнуть весь избыточный поток из y. Показаны только соседи y и ребра транспортной сети, которые входят в вершину y или покидают ее. В каждой части рисунка число внутри вершины представляет собой ее избыток в начале первой итерации, показанной в данной части; кроме того, в пределах части каждая вершина показана на своей высоте. Список соседей $y.N$ в начале каждой итерации приведен в правой части; в первой строке указан номер итерации. Заштрихованным соседом является $y.current$. **(а)** Изначально имеется 19 единиц избытка, которые должны быть протолкнуты из y, и $y.current = s$. Итерации 1–3 просто обновляют значение $y.current$, поскольку не имеется допустимых ребер, покидающих y. В итерации 4 $y.current = \text{NIL}$ (указано штриховкой под списком соседей), так что y поднимается и $y.current$ сбрасывается, получая в качестве значения заголовок списка соседей. **(б)** После подъема высота вершины y равна 1. В итерациях 5 и 6 выясняется, что ребра (y, s) и (y, x) недопустимые, но в итерации 7 выполняется проталкивание 8 единиц избыточного потока из y в z. Из-за проталкивания в этой итерации значение $y.current$ не изменяется. **(в)** Поскольку проталкивание в итерации 7 насыщает ребро (y, z), в итерации 8 обнаруживается его недопустимость. В итерации 9 $y.current = \text{NIL}$, так что вершина y снова поднимается, а $y.current$ сбрасывается.

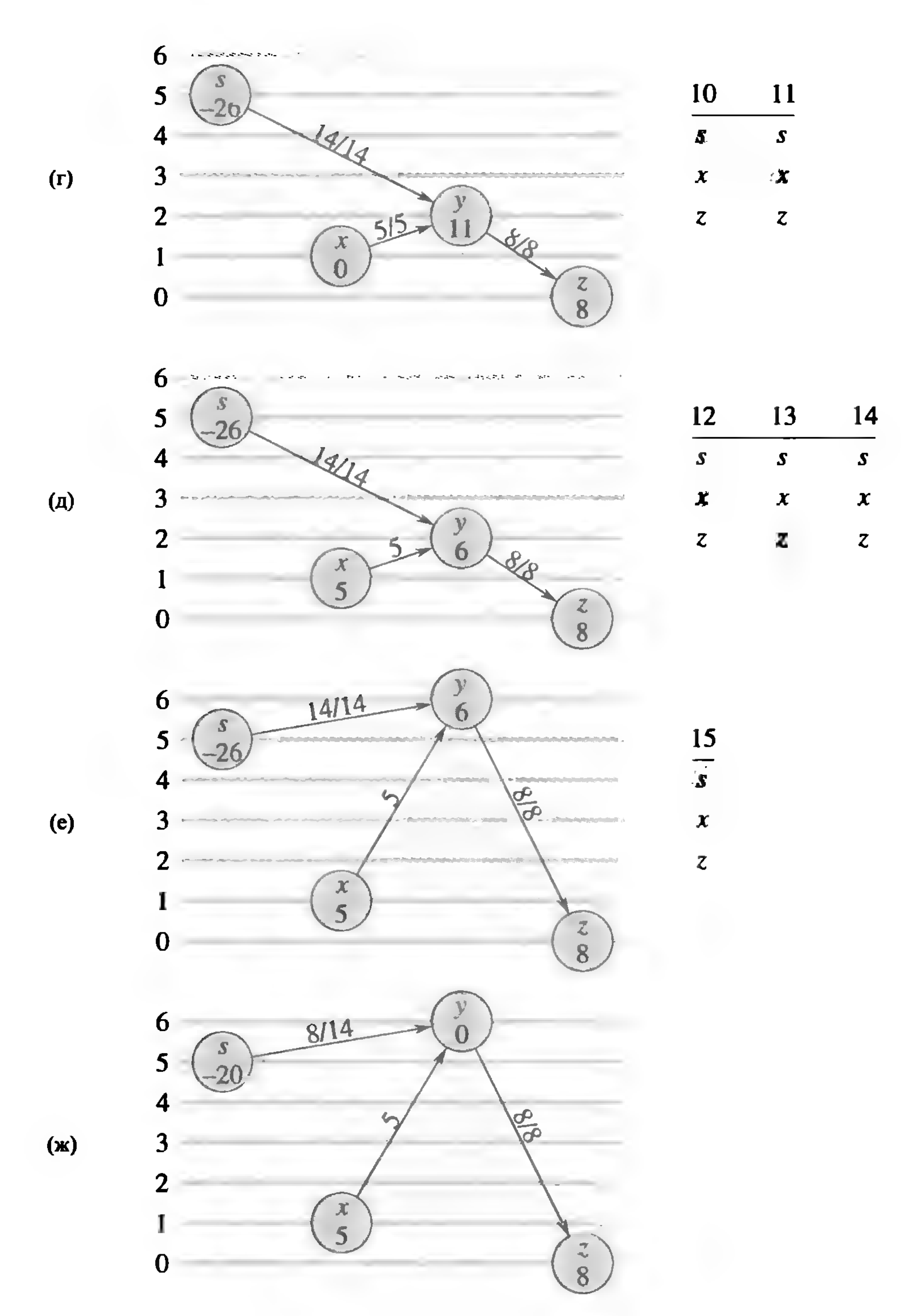


Рис. 26.9 (продолжение). **(г)** В итерации 10 ребро (y, s) недопустимо, но итерация 11 проталкивает 5 единиц избыточного потока из y в x. **(д)** Поскольку значение $y.current$ не изменялось в итерации 11, в итерации 12 обнаруживается, что ребро (y, x) недопустимое. Итерация 13 находит недопустимость ребра (y, z), а итерация 14 поднимает вершину y и сбрасывает значение $y.current$. **(е)** Итерация 15 проталкивает 6 единиц избыточного потока из y в s. **(ж)** Вершина y теперь не имеет избыточного потока, и процедура DISCHARGE завершается. В этом примере процедура DISCHARGE и начинается, и завершается с текущим указателем на заголовок списка соседей, но в общем случае это не обязательно так.

Чтобы доказать второе утверждение, исходя из проверки в строке 1 и леммы 26.28, необходимо только показать, что все ребра, выходящие из u, являются недопустимыми. Если вызов DISCHARGE(u) начинается с указателем $u.current$ на голову списка соседей, а по завершении он указывает за конец списка, то все выходящие из u ребра недопустимы, и применяется операция подъема. Возможно, однако, что во время вызова DISCHARGE(u) указатель $u.current$ проходит только по части списка перед возвратом из процедуры. После этого могут произойти вызовы процедуры DISCHARGE с другими вершинами, но указатель $u.current$ будет перемещаться по списку в процессе следующего вызова DISCHARGE(u). Рассмотрим теперь, что произойдет при полном проходе по списку, который начинается с заголовка $u.N$ и заканчивается значением $u.current =$ NIL. Когда $u.current$ достигает конца списка, процедура поднимает u и начинает новый проход. Чтобы в процессе прохода указатель $u.current$ переместился за вершину $v \in u.N$, ребро (u, v) должно быть признано недопустимым проверкой в строке 6. Таким образом, к моменту завершения прохода каждое ребро, покидающее u, определено как недопустимое в некоторый момент этого прохода. Ключевым является тот факт, что к концу прохода все ребра, покидающие u, остаются недопустимыми. Почему? Согласно лемме 26.27 операции проталкивания не могут приводить к созданию допустимых ребер, независимо от того, из какой вершины выполняется проталкивание. Таким образом, любое допустимое ребро должно быть создано операцией подъема. Но вершина u не подвергается подъему в процессе прохода, а любая другая вершина v, подвергшаяся подъему в процессе данного прохода (в результате вызова DISCHARGE(v)), не имеет после подъема допустимых входящих ребер согласно лемме 26.28. Итак, в конце прохода все ребра, выходящие из u, остаются недопустимыми, и лемма доказана. ■

Алгоритм "поднять-в-начало"

В алгоритме "поднять-в-начало" поддерживается связанный список L, состоящий из всех вершин множества $V - \{s, t\}$. Ключевым свойством данного списка является то, что вершины в нем топологически отсортированы в соответствии с допустимой сетью, как будет показано при рассмотрении инварианта цикла ниже. (Напомним, что согласно лемме 26.26 допустимая сеть является ориентированным ациклическим графом.)

В приведенном ниже псевдокоде алгоритма "поднять-в-начало" предполагается, что для каждой вершины u уже создан список соседей $u.N$. Кроме того, предполагается, что $u.next$ указывает на вершину, следующую за u в списке L, и что, как обычно, если u — последняя вершина данного списка, то $u.next =$ NIL.

```
RELABEL-TO-FRONT(G, s, t)
 1  INITIALIZE-PREFLOW(G, s)
 2  L = G.V − {s, t}, в произвольном порядке
 3  for каждой вершины u ∈ G.V − {s, t}
 4      u.current = u.N.head
 5  u = L.head
 6  while u ≠ NIL
 7      old-height = u.h
 8      DISCHARGE(u)
 9      if u.h > old-height
10          переместить u в начало списка L
11      u = u.next
```

Алгоритм "поднять-в-начало" работает следующим образом. Строка 1 инициализирует предпоток и высоты теми же значениями, что и обобщенный алгоритм проталкивания предпотока. Строка 2 инициализирует список L, который содержит все потенциально переполненные вершины в произвольном порядке. Строки 3 и 4 инициализируют указатель *current* каждой вершины u таким образом, чтобы он указывал на первую вершину в списке соседей u.

Как показано на рис. 26.10, цикл **while** (строки 6–11) проходит по списку L, разгружая вершины. Рассмотрение начинается с первой вершины списка L (строка 5). На каждой итерации цикла выполняется разгрузка вершины u (строка 8). Если процедура DISCHARGE изменила высоту вершины u, строка 10 перемещает эту вершину в начало списка L. Чтобы определить, подверглась ли вершина u подъему, перед разгрузкой ее высота сохраняется в переменной *old-height* (строка 7), а затем это значение сравнивается со значением высоты после выполнения процедуры разгрузки (строка 9). Строка 11 обеспечивает выполнение очередной итерации цикла **while** для вершины, следующей за u в списке L. Если u была передвинута в начало списка в строке 10, рассматриваемая на следующей итерации вершина представляет собой вершину, следующую за u в ее новой позиции в списке.

Чтобы показать, что процедура RELABEL-TO-FRONT вычисляет максимальный поток, покажем, что она является реализацией обобщенного алгоритма проталкивания предпотока. Во-первых, заметим, что она выполняет операции проталкивания и подъема только тогда, когда они применимы, что гарантируется леммой 26.29. Остается показать, что, когда процедура RELABEL-TO-FRONT завершается, не применима ни одна основная операция. Дальнейшее доказательство корректности построено на следующем инварианте цикла:

> При каждом выполнении проверки в строке 6 процедуры RELABEL-TO-FRONT список L является топологическим упорядочением вершин допустимой сети $G_{f,h} = (V, E_{f,h})$, и ни одна вершина, стоящая в списке перед u, не имеет избыточного потока.

Инициализация. Непосредственно после запуска процедуры INITIALIZE-PREFLOW $s.h = |V|$ и $v.h = 0$ для всех $v \in V - \{s\}$. Поскольку $|V| \geq 2$

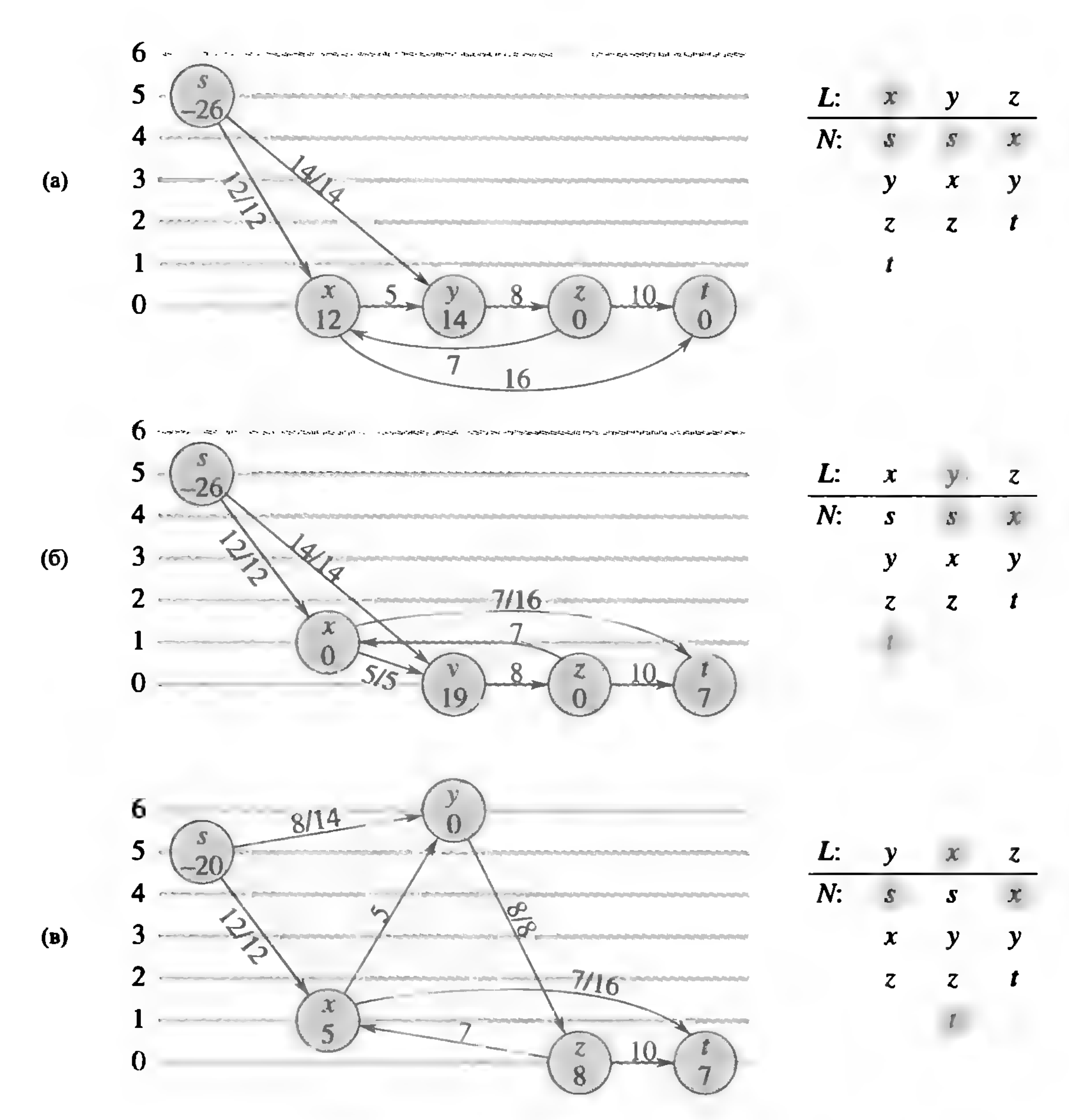


Рис. 26.10. Работа процедуры RELABEL-TO-FRONT. **(а)** Транспортная сеть непосредственно перед первой итерацией цикла **while**. Изначально источник s покидают 26 единиц потока. В правой части показан исходный список $L = \langle x, y, z \rangle$, где изначально $u = x$. Под каждой вершиной в списке L приведен ее список соседей, в котором выделен текущий сосед. Он поднимается до высоты 1, 5 единиц избыточного потока проталкиваются в y, а 7 оставшихся единиц избытка проталкиваются в сток t. Поскольку x поднята, она перемещается в начало списка L, что в данном случае не приводит к изменению структуры L. **(б)** После x следующей в L вершиной, подвергающейся разгрузке, является вершина y. На рис. 26.9 детально показан процесс разгрузки y в этой ситуации. Поскольку вершина y поднята, она перемещается в начало списка L. **(в)** Теперь вершина x следует в списке L за y, так что она вновь разгружается с проталкиванием всех 5 единиц избытка в t. Поскольку вершина x в этой операции разгрузки не поднимается, она остается на своем месте в списке L.

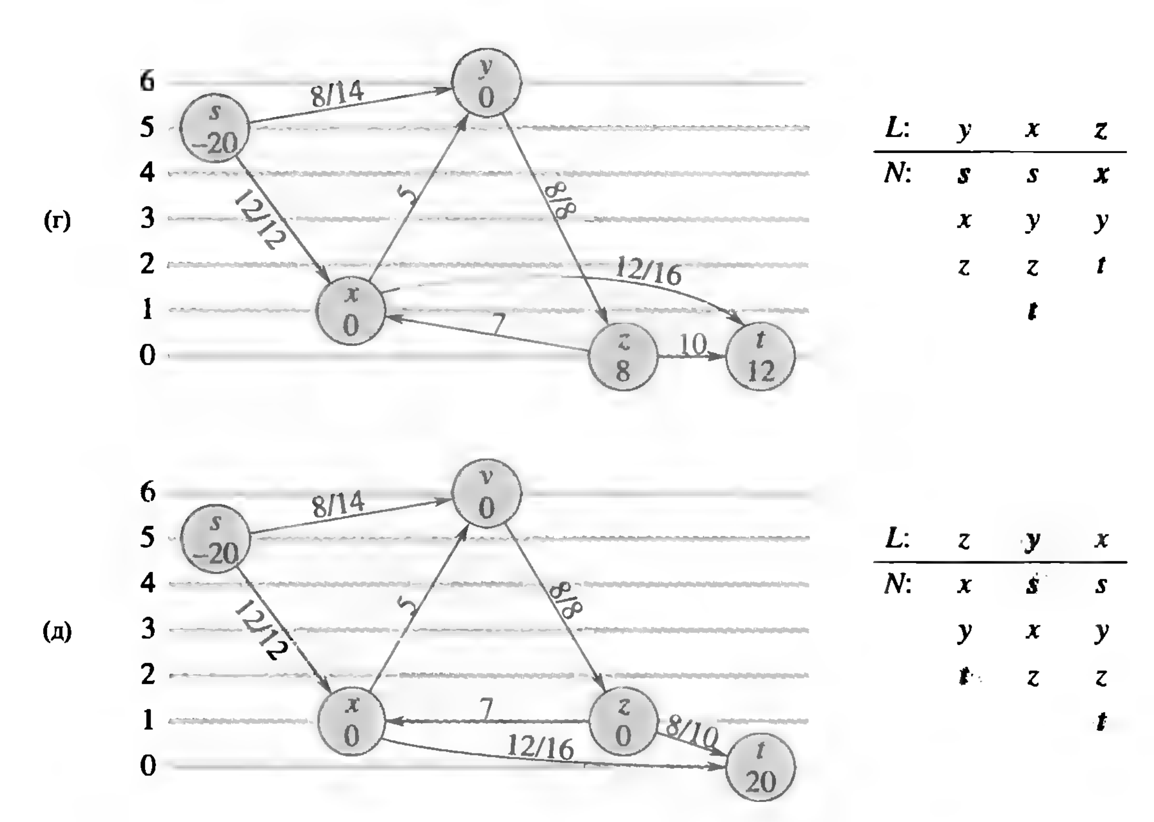


Рис. 26.10 (продолжение). **(г)** Поскольку в списке L за вершиной x следует вершина z, она подвергается разгрузке. Эта вершина поднимается до высоты 1, а все 8 единиц избытка потока проталкиваются в t. Поскольку z поднята, она перемещается в начало списка L. **(д)** Теперь за вершиной z в списке L следует вершина y, так что должна быть выполнена ее разгрузка. Но поскольку в вершине y избыток отсутствует, процедура DISCHARGE тут же выполняет возврат, и y остается в L на своем месте. Затем выполняется разгрузка вершины x. Поскольку она также не имеет избытка, процедура DISCHARGE вновь выполняет немедленный возврат, и x также остается на собственном месте в L. Процедура RELABEL-TO-FRONT достигает конца списка L и завершается. Переполненных вершин нет, и предпоток представляет собой максимальный поток.

(так как V содержит как минимум исток s и сток t), ни одно ребро не является допустимым. Следовательно, $E_{f,h} = \emptyset$, и любое упорядочение множества $V - \{s, t\}$ является топологическим упорядочением $G_{f,h}$.

Поскольку изначально вершина u является заголовком списка L, перед ней нет вершин, следовательно, перед ней нет вершин с избытком потока.

Сохранение. Чтобы убедиться в том, что данное топологическое упорядочение сохраняется при проведении итераций цикла **while**, прежде всего заметим, что к изменению допустимой сети приводят только операции проталкивания и подъема. Согласно лемме 26.27 операции проталкивания не приводят к тому, что ребра становятся допустимыми. Поэтому допустимые ребра могут создаваться только подъемами. Однако после того как вершина u подверглась подъему, согласно лемме 26.28 больше не существует допустимых ребер, входящих в u, но могут быть допустимые ребра, выходящие из нее. Таким образом, перемещая u в начало списка L, алгоритм гарантирует, что все допустимые ребра, выходящие из u, удовлетворяют условию топологического упорядочения.

Чтобы убедиться в том, что ни одна вершина, предшествующая u в списке L, не имеет избытка потока, обозначим вершину, которая будет текущей вершиной u на следующей итерации, как u'. Среди вершин, которые будут предшествовать u' на следующей итерации, находится текущая вершина u (согласно строке 11), и либо больше таких вершин нет (если u подвергалась подъему), либо там находятся те же вершины, что и ранее (если u не поднималась). Поскольку u подверглась разгрузке, она не содержит избытка потока. Следовательно, если u подвергалась подъему в процессе разгрузки, то ни одна вершина, предшествующая u', не содержит избытка потока. Если же u в процессе разгрузки не поднималась, ни одна вершина, стоящая в списке L перед ней, не получила избыток потока при этой разгрузке, поскольку список L остается топологически упорядоченным все время в процессе разгрузки (как уже отмечалось, допустимые ребра создаются только подъемами, а не операциями проталкивания), поэтому каждая операция проталкивания заставляет избыток потока двигаться только к вершинам, расположенным в списке дальше (или же к s или t). И вновь ни одна вершина, предшествующая u', не имеет избытка потока.

Завершение. Когда цикл завершается, u оказывается за последним элементом списка L, поэтому инвариант цикла гарантирует, что избыток всех вершин равен 0. Следовательно, ни одна основная операция неприменима.

Анализ

Покажем теперь, что процедура RELABEL-TO-FRONT выполняется за время $O(V^3)$ для любой транспортной сети $G = (V, E)$. Поскольку данный алгоритм является реализацией обобщенного алгоритма проталкивания предпотока, воспользуемся следствием 26.21, которое устанавливает границу $O(V)$ для числа подъемов, применяемых к одной вершине, и $O(V^2)$ для общего числа подъемов. Кроме того, в упр. 26.4.3 устанавливается граница $O(VE)$ для суммарного времени, затраченного на выполнение подъемов, а лемма 26.22 устанавливает границу $O(VE)$ для суммарного числа операций насыщающих проталкиваний.

Теорема 26.30

Время выполнения процедуры RELABEL-TO-FRONT для любой транспортной сети $G = (V, E)$ составляет $O(V^3)$.

Доказательство. Будем считать “фазой” алгоритма “поднять-в-начало” время между двумя последовательными операциями подъема. Поскольку всего выполняется $O(V^2)$ подъемов, в алгоритме насчитывается $O(V^2)$ фаз. Каждая фаза сдержит не более $|V|$ вызовов процедуры DISCHARGE, что можно показать следующим образом. Если процедура DISCHARGE не выполняет подъем, то следующий ее вызов происходит ниже по списку L, а длина L меньше $|V|$. Если же процедура DISCHARGE выполняет подъем, то следующий ее вызов происходит уже в другой фазе алгоритма. Поскольку каждая фаза содержит не более $|V|$ обращений к процедуре DISCHARGE, а всего в алгоритме насчитывается $O(V^2)$ фаз, число вызовов данной процедуры в строке 8 процедуры RELABEL-TO-FRONT составляет $O(V^3)$.

Таким образом, цикл **while** процедуры RELABEL-TO-FRONT выполняет работу (не учитывая работу, выполняемую внутри процедуры DISCHARGE), не превышающую $O(V^3)$.

Теперь необходимо проанализировать работу процедуры DISCHARGE в ходе выполнения данного алгоритма. Каждая итерация **while** в процедуре DISCHARGE заключается в выполнении одного из трех действий. Проанализируем объем работы при выполнении каждого из них.

Начнем с подъемов (строки 4 и 5). В упр. 26.4.3 время выполнения всех $O(V^2)$ подъемов ограничивается пределом $O(VE)$.

Теперь предположим, что действие заключается в обновлении указателя $u.current$ в строке 8. Это действие выполняется $O(\text{degree}(u))$ раз всякий раз, когда вершина u подвергается подъему, что в целом для вершины составляет $O(V \cdot \text{degree}(u))$ раз. Следовательно, для всех вершин общий объем работы по перемещению указателей в списках соседей составляет $O(VE)$ согласно лемме о рукопожатиях (упр. Б.4.1).

Третий тип действий, выполняемых процедурой DISCHARGE, — операция проталкивания (строка 7). Мы уже знаем, что суммарное число насыщающих операций проталкивания составляет $O(VE)$. Заметим, что если выполняется ненасыщающее проталкивание, процедура DISCHARGE немедленно выполняет возврат, поскольку такое проталкивание уменьшает избыток до 0. Поэтому при каждом обращении к процедуре DISCHARGE может выполняться не более одного ненасыщающего проталкивания. Как мы знаем, процедура DISCHARGE вызывается $O(V^3)$ раз, следовательно, общее время, затраченное на ненасыщающие проталкивания, составляет $O(V^3)$.

Таким образом, время выполнения процедуры RELABEL-TO-FRONT составляет $O(V^3 + VE)$, что эквивалентно $O(V^3)$. ■

Упражнения

26.5.1

Проиллюстрируйте, используя в качестве образца рис. 26.10, выполнение процедуры RELABEL-TO-FRONT для транспортной сети, представленной на рис. 26.1, (а). Предполагается, что начальный порядок следования вершин в списке L — $\langle v_1, v_2, v_3, v_4 \rangle$, а списки соседей имеют следующий вид:

$$\begin{aligned} v_1.N &= \langle s, v_2, v_3 \rangle , \\ v_2.N &= \langle s, v_1, v_3, v_4 \rangle , \\ v_3.N &= \langle v_1, v_2, v_4, t \rangle , \\ v_4.N &= \langle v_2, v_3, t \rangle . \end{aligned}$$

26.5.2 ★

Необходимо реализовать алгоритм проталкивания предпотока, в котором поддерживается порядок обслуживания переполненных вершин "первым вошел, первым вышел". Данный алгоритм разгружает первую вершину в очереди и удаляет ее оттуда, а все вершины, которые перед этой разгрузкой не были переполнены, но по-

сле нее стали таковыми, помещаются в конец очереди. Когда очередь становится пустой, алгоритм завершается. Покажите, что можно построить реализацию данного алгоритма, которая вычисляет максимальный поток за время $O(V^3)$.

26.5.3

Покажите, что обобщенный алгоритм будет работать, даже если процедура RELABEL обновляет $u.h$, просто вычисляя $u.h = u.h + 1$. Как это повлияет на оценку времени выполнения процедуры RELABEL-TO-FRONT?

26.5.4 ★

Покажите, что если разгрузке всегда подвергается наивысшая переполненная вершина, метод проталкивания предпотока можно реализовать так, чтобы он выполнялся за время $O(V^3)$.

26.5.5

Предположим, что в некоторой точке в процессе выполнения алгоритма проталкивания предпотока существует некоторое целое число $0 < k \leq |V| - 1$, для которого нет ни одной вершины с данной высотой, т.е. такой, у которой $v.h = k$. Покажите, что все вершины с $v.h > k$ находятся в минимальном разрезе на стороне истока. Если такое значение k существует, ***эвристика промежутка*** (gap heuristic) обновляет каждую вершину $v \in V - \{s\}$, для которой $v.h > k$, устанавливая $v.h = \max(v.h, |V| + 1)$. Покажите, что полученный таким образом атрибут h является функцией высоты. (Эвристика промежутка позволяет получить эффективные реализации метода проталкивания предпотока на практике.)

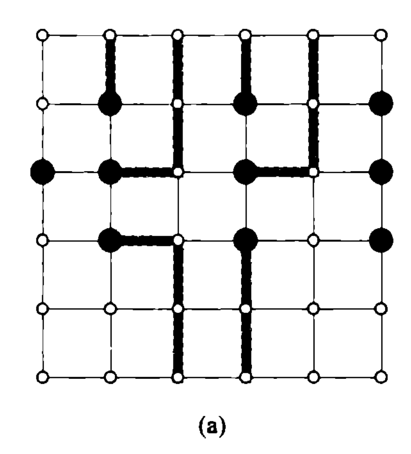
Задачи

26.1. Задача о выходе

Решетка (grid) размером $n \times n$ представляет собой неориентированный граф, состоящий из n строк и n столбцов вершин, как показано на рис. 26.11. Обозначим вершину, находящуюся в i-й строке и j-м столбце, как (i, j). Каждая вершина решетки имеет ровно по четыре соседа, за исключением граничных вершин, представляющих собой точки (i, j), для которых $i = 1$, $i = n$, $j = 1$ или $j = n$.

Пусть в решетке задано $m \leq n^2$ стартовых точек (x_1, y_1), (x_2, y_2), ..., (x_m, y_m). ***Задача о выходе*** (escape problem) заключается в том, чтобы определить, существует ли m путей, не имеющих общих вершин, из стартовых точек к любым m различным точкам границы. Например, решетка на рис. 26.11, (а) имеет выход, а решетка на рис. 26.11, (б) — не имеет.

а. Рассмотрим транспортную сеть, в которой вершины, равно как и ребра, имеют пропускные способности, т.е. суммарный положительный поток, входящий в каждую заданную вершину, должен удовлетворять ограничению пропускной способности. Покажите, что задача определения максимального потока в такой сети, где вершины и ребра имеют пропускные способности, может быть



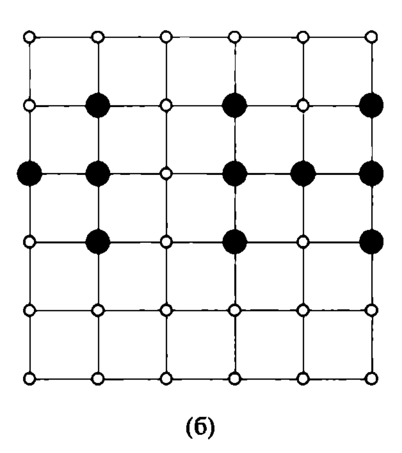


Рис. 26.11. Решетки для задачи о выходе. Стартовые точки — черные, прочие вершины решетки — белые. **(а)** Решетка с выходом, представленным заштрихованными путями. **(б)** Решетка без выхода.

сведена к стандартной задаче о максимальном потоке для транспортной сети сопоставимого размера.

б. Разработайте эффективный алгоритм решения задачи о выходе и проанализируйте время его выполнения.

26.2. Задача о минимальном покрытии путями

Покрытие путями (path cover) ориентированного графа $G = (V, E)$ — это множество P не имеющих общих вершин путей, таких, что каждая вершина из множества V принадлежит ровно одному пути из P. Пути могут начинаться и заканчиваться где угодно, а также иметь произвольную длину, включая 0. ***Минимальным покрытием путями*** (minimum path cover) графа G называется покрытие, содержащее наименьшее возможное количество путей.

а. Предложите эффективный алгоритм поиска минимального покрытия путями ориентированного ациклического графа $G = (V, E)$. (*Указание:* предположив, что $V = \{1, 2, \ldots, n\}$, постройте граф $G' = (V', E')$, где

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} \ ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} \ ,$$

и примените алгоритм поиска максимального потока.)

б. Будет ли ваш алгоритм работать для ориентированного графа, содержащего циклы? Объясните свой ответ.

26.3. Алгоритмическое консультирование

Профессор решил открыть компанию, проводящую консультации по разным алгоритмам. Он разделил алгоритмы на n важных областей (грубо соответствующих различным частям этой книги), представленных множеством $A = \{A_1, A_2, \ldots, A_n\}$. Для каждой подобласти A_k он может нанять эксперта по данной тематике за c_k долларов. У компании имеется множество $J = \{J_1, J_2, \ldots, J_m\}$ потенциальных задач. Для выполнения задачи J_i компания должна нанять экспертов

для подмножества $R_i \subseteq A$ подобластей. Каждый эксперт может одновременно работать над несколькими задачами. Если компания решает взяться за задачу J_i, она должна нанять экспертов во всех подобластях в R_i, и получит за выполнение задачи оплату в p_i долларов.

Профессору необходимо определить, для каких подобластей следует нанять экспертов и за какие задачи взяться, чтобы максимизировать чистую прибыль, равную плате за выполненные задачи минус оплата труда нанятых экспертов.

Рассмотрим следующую транспортную сеть G. Она содержит исток s, вершины $A_1, A_2, \ldots, A_n$, вершины $J_1, J_2, \ldots, J_m$ и сток t. Транспортная сеть содержит ребра (s, A_k) ($k = 1, 2 \ldots, n$) с пропускной способностью $c(s, A_k) = c_k$, и для $i = 1, 2, \ldots, m$ транспортная сеть содержит ребра (J_i, t) с пропускной способностью $c(J_i, t) = p_i$. Если $A_k \in R_i$ ($k = 1, 2, \ldots, n$ и $i = 1, 2, \ldots, m$), то G содержит ребро (A_k, J_i) с пропускной способностью $c(A_k, J_i) = \infty$.

а. Покажите, что если $J_i \in T$ для разреза (S, T) с конечной пропускной способностью транспортной сети G, то $A_k \in T$ для каждого $A_k \in R_i$.

б. Покажите, как определить максимальную чистую прибыль из пропускной способности минимального разреза транспортной сети G и заданных значений p_i.

в. Разработайте эффективный алгоритм определения того, за какие задачи следует взяться и каких экспертов нанять. Найдите время работы своего алгоритма как функцию от m, n и $r = \sum_{i=1}^{m} |R_i|$.

26.4. Обновление максимального потока

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t и целочисленными пропускными способностями. Предположим, что известен максимальный поток в G.

а. Предположим, что пропускная способность некоторого одного ребра $(u, v) \in E$ увеличена на 1. Предложите алгоритм обновления максимального потока с временем выполнения $O(V + E)$.

б. Предположим, что пропускная способность некоторого одного ребра $(u, v) \in E$ уменьшена на 1. Предложите алгоритм обновления максимального потока с временем выполнения $O(V + E)$.

26.5. Масштабирование

Пусть $G = (V, E)$ представляет собой транспортную сеть с истоком s и стоком t и целочисленными пропускными способностями $c(u, v)$ каждого ребра $(u, v) \in E$. Пусть $C = \max_{(u,v)\in E} c(u, v)$.

а. Докажите, что минимальный разрез G имеет пропускную способность не более $C\,|E|$.

б. Для заданного числа K покажите, как за время $O(E)$ можно найти увеличивающий путь с пропускной способностью не менее K, если таковой путь существует.

Для вычисления максимального потока в G можно использовать следующую модификацию процедуры FORD-FULKERSON-METHOD.

MAX-FLOW-BY-SCALING(G, s, t)
1 $C = \max_{(u,v) \in E} c(u, v)$
2 Инициализировать поток f значением 0
3 $K = 2^{\lfloor \lg C \rfloor}$
4 **while** $K \geq 1$
5   **while** существует увеличивающий путь p
            с пропускной способностью не менее K
6   Увеличить поток f вдоль p
7     $K = K/2$
8 **return** f

в. Докажите, что MAX-FLOW-BY-SCALING возвращает максимальный поток.

г. Покажите, что пропускная способность минимального разреза остаточной сети G_f не превышает $2K\,|E|$ при каждом выполнении строки 4.

д. Докажите, что внутренний цикл **while** в строках 5 и 6 выполняется $O(E)$ раз для каждого значения K.

е. Сделайте вывод о том, что процедуру MAX-FLOW-BY-SCALING можно реализовать таким образом, что она будет выполняться за время $O(E^2 \lg C)$.

26.6. Алгоритм Хопкрофта–Карпа поиска паросочетания в двудольном графе

В данной задаче представлен более быстрый алгоритм поиска максимального паросочетания в двудольном графе, предложенный Хопкрофтом (Hopcroft) и Карпом (Karp). Этот алгоритм выполняется за время $O(\sqrt{V}E)$. Задан неориентированный двудольный граф $G = (V, E)$, где $V = L \cup R$ и у всех ребер ровно одна конечная точка находится в L. Пусть M — паросочетание в G. Мы говорим, что простой путь P в G является ***увеличивающим путем*** (augmenting path) по отношению к M, если он начинается в некоторой свободной вершине множества L, заканчивается в некоторой свободной вершине R, а его ребра попеременно принадлежат M и $E - M$. (Это определение увеличивающего пути связано с определением увеличивающего пути в транспортной сети, но несколько отличается от него.) В данной задаче путь трактуется как последовательность ребер, а не последовательность вершин. Кратчайший увеличивающий путь по отношению к паросочетанию M — это увеличивающий путь с минимальным числом ребер.

Для заданных двух множеств A и B ***симметрическая разность*** (symmetric difference) $A \oplus B$ определяется как $(A - B) \cup (B - A)$, т.е. это элементы, которые

а. Покажите, что если M представляет собой некоторое паросочетание, а P — увеличивающий путь по отношению к M, то симметрическая разность множеств $M \oplus P$ является паросочетанием, и $|M \oplus P| = |M| + 1$. Покажите, что если $P_1, P_2, \ldots, P_k$ — увеличивающие пути по отношению к M, не имеющие общих вершин, то симметрическая разность $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ является паросочетанием с мощностью $|M| + k$.

Общая структура алгоритма имеет следующий вид.

```
HOPCROFT-KARP(G)
1  M = ∅
2  repeat
3      Пусть 𝒫 = {P1, P2, ..., Pk} — максимальное множество
           кратчайших увеличивающих путей по отношению
           к M, не имеющих общих вершин
4      M = M ⊕ (P1 ∪ P2 ∪ ··· ∪ Pk)
5  until 𝒫 == ∅
6  return M
```

Далее в этой задаче вам предлагается проанализировать число итераций данного алгоритма (т.е. число итераций цикла **repeat**) и предложить реализацию строки 3.

б. Для двух заданных паросочетаний M и M^* в G покажите, что каждая вершина графа $G' = (V, M \oplus M^*)$ имеет степень не больше 2. Сделайте вывод, что G' является несвязным объединением простых путей или циклов. Докажите, что ребра каждого такого простого пути или цикла по очереди принадлежат M и M^*. Докажите, что если $|M| \leq |M^*|$, то $M \oplus M^*$ содержит как минимум $|M^*| - |M|$ увеличивающих путей по отношению к M, не имеющих общих вершин.

Пусть l обозначает длину кратчайшего увеличивающего пути по отношению к паросочетанию M и пусть $P_1, P_2, \ldots, P_k$ представляет собой максимальное множество не имеющих общих вершин увеличивающих путей длиной l по отношению к M. Пусть $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, и предположим, что P — кратчайший увеличивающий путь по отношению к M'.

в. Покажите, что если P не имеет общих вершин с $P_1, P_2, \ldots, P_k$, то P содержит более l ребер.

г. Теперь предположим, что P может иметь общие вершины с $P_1, P_2, \ldots, P_k$. Пусть A — множество ребер $(M \oplus M') \oplus P$. Покажите, что $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ и что $|A| \geq (k + 1)l$. Сделайте вывод о том, что P содержит более l ребер.

д. Докажите, что если кратчайший увеличивающий путь для M содержит l ребер, то размер максимального паросочетания составляет не более $|M| + |V| / (l + 1)$.

е. Покажите, что число повторений цикла **repeat** в данном алгоритме не превышает $2\sqrt{|V|}$. (*Указание:* насколько сможет вырасти M после итерации номер $\sqrt{|V|}$?)

ж. Предложите алгоритм для поиска максимального множества не имеющих общих вершин кратчайших увеличивающих путей $P_1, P_2, \ldots, P_k$ для заданного паросочетания M, время работы которого составляет $O(E)$. Заключите отсюда, что суммарное время выполнения процедуры HOPCROFT-KARP составляет $O(\sqrt{V}E)$.

Заключительные замечания

Транспортные сети и связанные с ними алгоритмы рассматриваются в работах Ахуя (Ahuja), Магнанти (Magnanti) и Орлина (Orlin) [7], Ивена (Even) [102], Лоулера (Lawler) [223], Пападимитриу (Papadimitriou) и Стейглица (Steiglitz) [269], Таржана (Tarjan) [328]. Широкий обзор алгоритмов для задач поиска потоков в транспортных сетях можно найти также в книге Голдберга (Goldberg), Тардоса (Tardos) и Таржана [138]. В работе Шрайвера (Schrijver) [302] предлагается интересный исторический обзор исследований в сфере транспортных сетей.

Метод Форда–Фалкерсона представлен в работе Форда (Ford) и Фалкерсона (Fulkerson) [108], которые являются основоположниками формальных исследований ряда задач в области транспортных сетей, включая задачи поиска максимального потока и паросочетаний. Во многих ранних реализациях метода Форда–Фалкерсона поиск увеличивающих путей осуществляется с помощью поиска в ширину; Эдмондс (Edmonds) и Карп (Karp) [101] (и независимо от них Диниц (Dinic) [88]) доказали, что такая стратегия дает полиномиальный по времени алгоритм. Диницу [88] также принадлежит идея использования "тупиковых потоков" (blocking flows); предпотоки впервые предложил Карзанов (Karzanov) [201]. Метод проталкивания предпотока описан в работах Голдберга [135] и Голдберга и Таржана [139]. Голдберг и Таржан приводят алгоритм со временем работы $O(V^3)$, в котором для хранения множества переполненных вершин используется очередь, а также алгоритм на основе использования динамических деревьев, время работы которого достигает $O(VE\lg(V^2/E+2))$. Некоторые другие исследователи разработали алгоритмы проталкивания предпотока для поиска максимального потока. В работах Ахуя и Орлина [9] и Ахуя, Орлина и Таржана [10] приводятся алгоритмы, использующие масштабирование. Чериян (Cheriyan) и Махешвари (Maheshvari) [61] предложили проталкивать поток из переполненной вершины с максимальной высотой. В работе Черияна и Хейджерапа (Hagerup) [60] предлагается использовать случайные перестановки списков соседей; другие исследователи [14, 203, 274] развили данную идею, предложив искусные методы дерандомизации, что позволило получить ряд более быстрых алгоритмов. Алгоритм, предложенный Кингом (King), Рао (Rao) и Таржаном [203], является самым быстрым из них — время его работы составляет $O(VE\log_{E/(V\lg V)} V)$.

Асимптотически самый быстрый из известных в настоящее время алгоритмов для задачи максимального потока разработан Голдбергом и Рао [137], время его работы равно $O(\min(V^{2/3}, E^{1/2})E\lg(V^2/E+2)\lg C)$, где $C = \max_{(u,v)\in E} c(u,v)$. Этот алгоритм не использует метод проталкивания предпотока, он основан на нахождении тупиковых потоков. Все предыдущие алгоритмы, включая рассмотренные в данной главе, используют некоторое понятие расстояния (в алгоритмах проталкивания предпотока используется аналогичное понятие высоты), где каждому ребру неявно присвоена длина 1. В этом же алгоритме используется другой подход: ребрам с высокой пропускной способностью присваивается длина 0, а ребрам с низкой пропускной способностью — длина 1. Неформально при таком выборе длин кратчайшие пути от истока к стоку будут иметь высокую пропускную способность, следовательно, потребуется меньше итераций.

На практике на сегодняшний день при решении задач поиска максимального потока алгоритмы проталкивания предпотока превосходят алгоритмы, основанные на увеличивающих путях и линейном программировании. В исследованиях Черкасски (Cherkassky) и Голдберга [62] подчеркивается важность использования при реализации алгоритма проталкивания предпотока двух эвристик. Первая состоит в том, что в остаточной сети периодически выполняется поиск в ширину, чтобы получить более точные значения высот. Вторая эвристика — это "эвристика промежутка" (gap heuristic), описанная в упр. 26.5.5. Авторы пришли к заключению, что наилучшим вариантом метода проталкивания предпотока является вариант, в котором для разгрузки выбирается переполненная вершина с максимальной высотой.

Наилучший известный к настоящему времени алгоритм поиска максимального паросочетания (описанный в задаче 26.6) был предложен Хопкрофтом (Hopcroft) и Карпом (Karp) [175]; время его работы составляет $O(\sqrt{V}E)$. Задачи поиска паросочетаний подробно рассматриваются в книге Ловаса (Lovász) и Пламмера (Plummer) [238].



VII Избранные темы

Введение

В этой части содержатся избранные темы теории алгоритмов, расширяющие и дополняющие материал, изложенный в данной книге ранее. В одних главах вводятся новые вычислительные модели, такие как комбинационные схемы или параллельные вычислительные машины. Другие главы охватывают специализированные области знаний, такие как вычислительная геометрия или теория чисел. В двух последних главах обсуждаются некоторые известные ограничения, возникающие при разработке эффективных алгоритмов, а также излагаются основы методов, позволяющих справиться с этими ограничениями.

В главе 27 представлена алгоритмическая модель для параллельных вычислений на основе динамической многопоточности. В этой главе сначала вводятся основы указанной модели и демонстрируется, как количественно описать параллелизм, а затем изучается несколько интересных многопоточных алгоритмов, включая алгоритмы умножения матриц и сортировки слиянием.

В главе 28 изучаются эффективные алгоритмы, предназначенные для работы с матрицами. В ней представлены два общих метода — LU-разложение и LUP-разложение. Они предназначены для решения системы линейных уравнений по методу исключения Гаусса за время $O(n^3)$. Здесь также показано, что перемножение и обращение матриц можно выполнять с одинаковой скоростью. В заключительной части главы показано, как получить приближенное решение системы линейных уравнений методом наименьших квадратов, если эта система не имеет точного решения.

В главе 29 исследуется линейное программирование, цель которого — минимизировать или максимизировать целевую функцию при заданных ограниченных ресурсах и конкурирующих ограничениях. Линейное программирование применяется в самых различных прикладных областях. В этой главе описывается постановка задач линейного программирования и их решение. В качестве метода решения предложен симплекс-алгоритм, который является одним из древнейших алгоритмов, используемых в линейном программировании. В отличие от многих

других алгоритмов, о которых идет речь в этой книге, для симплекс-алгоритма время работы в наихудшем случае не выражается полиномиальной функцией, однако он достаточно эффективен и широко применяется на практике.

В главе 30 изучаются операции над полиномами. Здесь показано, как с помощью такого известного метода обработки сигналов, как быстрое преобразование Фурье (Fast Fourier Transform — FFT), можно перемножить два полинома n-й степени за время $O(n \lg n)$. В этой главе также исследуются методы эффективной реализации FFT, включая параллельные вычисления.

В главе 31 представлены теоретико-числовые алгоритмы. После обзора элементарной теории чисел здесь описан алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя. Далее представлены алгоритмы для решения модульных линейных уравнений и для возведения числа в степень по модулю другого числа. Затем читатель сможет ознакомиться с важным приложением теоретико-числовых алгоритмов: криптографической системой с открытым ключом RSA. С ее помощью можно не только кодировать сообщения таким образом, чтобы их не могли прочитать посторонние, но и создавать цифровые подписи. Далее в главе представлен рандомизированный тест простоты чисел Миллера–Рабина (Miller–Rabin), позволяющий выполнять эффективный поиск больших простых чисел, необходимый для реализации схемы RSA. В заключительной части главы описан эвристический ρ-метод Полларда (Pollard) для разбиения целых чисел на множители, а также обсуждаются успехи, достигнутые в области целочисленной факторизации.

В главе 32 исследуется задача поиска всех вхождений заданной строки-образца в имеющуюся строку текста; эта задача часто возникает при написании программ, предназначенных для редактирования текста. После ознакомления с "наивным" подходом в этой главе представлен элегантный метод решения данной задачи, разработанный Рабином (Rabin) и Карпом (Karp). Затем, после демонстрации эффективного решения, основанного на теории конечных автоматов, вниманию читателя предложен алгоритм Кнута–Морриса–Пратта (Knuth–Morris–Pratt), позволяющий достичь высокой эффективности за счет предварительной обработки образца.

Тема главы 33 — некоторые задачи вычислительной геометрии. После обсуждения основных примитивов этого раздела вычислительной математики в главе показано, как с помощью метода "обметания" можно эффективно определить, имеются ли пересечения в множестве прямолинейных отрезков. Два остроумных алгоритма, предназначенных для поиска выпуклой оболочки заданного множества точек — метод сканирования по Грэхему (Graham's scan) и метод продвижения по Джарвису (Jarvis's march), — также иллюстрируют мощь метода обметания. В заключение в главе описан эффективный алгоритм, предназначенный для поиска пары самых близких точек в заданном множестве точек на плоскости.

Глава 34 посвящена NP-полным задачам. Многие интересные вычислительные задачи являются NP-полными, однако неизвестен ни один алгоритм решения какой бы то ни было из этих задач, время работы которого выражалось бы полиномиальной функцией. В данной главе представлены методы определения того, является ли задача NP-полной. Доказана NP-полнота для нескольких классичес-

ких задач: определение того, содержит ли граф цикл Гамильтона, выполнима ли заданная булева формула и содержит ли заданное множество чисел такое подмножество, сумма элементов в котором была бы равна заданному значению. В этой главе, кроме того, доказано, что знаменитая задача о коммивояжере также является NP-полной.

В главе 35 показано, как эффективно находить приближенные решения NP-полных задач с помощью приближенных алгоритмов. Для одних NP-полных задач не так уж сложно выразить приближенные решения, достаточно близкие к оптимальным, в то время как для других задач даже самые лучшие из известных приближенных алгоритмов работают все хуже по мере увеличения размера задачи. Есть также другой класс задач, для которых наблюдается возрастание времени вычисления с увеличением точности приближенных решений. Эти возможности проиллюстрированы на примере решения задачи о вершинном покрытии (представлены невзвешенная и взвешенная версии), о коммивояжере и др.

Глава 27. Многопоточные алгоритмы

Подавляющее большинство алгоритмов в этой книге — ***последовательные*** (serial), пригодные для выполнения на однопроцессорном компьютере, который в каждый момент времени выполняет только одну инструкцию. В этой главе мы расширим нашу алгоритмическую модель ***параллельными алгоритмами*** (parallel algorithms), которые могут выполняться на многопроцессорных компьютерах, допускающих одновременное выполнение нескольких команд. В частности, мы рассмотрим элегантную модель динамических многопоточных алгоритмов, которые подчиняются общим принципам проектирования и анализа алгоритмов и при этом могут быть эффективно реализованы на практике.

Параллельные компьютеры, т.е. компьютеры с несколькими устройствами обработки данных, становятся все более распространенными и охватывают широкий диапазон цен и производительности. Относительно недорогие настольные и переносные компьютеры оснащены одним ***многоядерным*** процессором, в который входят несколько "ядер", которые сами по себе являются полноценными процессорами с доступом к общей памяти. В среднем диапазоне как по цене, так и по производительности находятся кластеры, составленные из отдельных компьютеров, зачастую относящихся к классу персональных, со связывающей их некоммутируемой сетью. К дорогостоящим относятся суперкомпьютеры, которые зачастую используют комбинацию специализированных архитектуры и сетей для достижения высочайшей производительности, выражаемой в количестве выполняемых за секунду команд.

В той или иной форме многопроцессорные компьютеры существуют уже десятилетия. Но несмотря на то, что модель машины с произвольным доступом для последовательных вычислений появилась и была принята еще на ранней стадии развития компьютерных наук, до сих пор ни одна модель для параллельных вычислений не получила широкого признания. Основная причина этого в том, что производители не договорились о единой архитектурной модели для параллельных компьютеров. Например, одни параллельные компьютеры оснащены ***совместно используемой памятью*** (shared memory), где каждый процессор может непосредственно обращаться к любой ячейке памяти. Другие параллельные компьютеры используют ***распределенную память*** (distributed memory), где каждый процессор имеет собственную память, и для доступа одного процессора к памяти другого между процессорами должны передаваться явные сообщения. Однако с появлением многоядерных технологий каждый новый настольный или перенос-

ной компьютер в настоящее время представляет собой параллельный компьютер с совместно используемой памятью. Время покажет, правы ли мы в своем выборе, но в данной главе принята именно эта модель — многопроцессорности с совместно используемой памятью.

Распространенным методом программирования многоядерных и иных параллельных компьютеров с совместно используемой памятью является применение ***статической многопоточности*** (static threading), предоставляющей программную абстракцию "виртуальных процессоров", или ***потоков*** (threads), совместно использующих общую память. Каждый поток поддерживает связанный с ним счетчик команд и может выполнять код независимо от других потоков. Операционная система загружает поток в процессор для выполнения и переключает потоки, когда выполнения требует другой поток. Хотя операционная система и позволяет программистам создавать и уничтожать потоки, эти операции относительно медленные. Таким образом, для большинства приложений потоки сохраняются на протяжении всего времени вычислений, почему они и получили название "статические".

К сожалению, непосредственное программирование параллельного компьютера с совместно используемой памятью с применением статических потоков — дело сложное и чреватое ошибками. Одна из причин этого заключается в сложности равномерного динамического распределения работы между потоками. Для любого (кроме самых простейших) приложения программист для сбалансированной загрузки потоков должен использовать сложные протоколы обмена информацией. Такое положение дел привело к созданию ***параллельных платформ*** (concurrency platforms), предоставляющих слой программного обеспечения, который координирует ресурсы параллельных вычислений, планирует их и управляет ими. Одни из таких платформ имеют вид библиотек времени выполнения, другие же предоставляют полноценные параллельные языки программирования с компиляторами и поддержкой времени выполнения.

Динамическое многопоточное программирование

Важным классом параллельных платформ является ***динамическая многопоточность*** (dynamic multithreading), которая представляет собой модель, используемую нами в данной главе. Динамическая многопоточность позволяет программисту указывать уровень параллелизма в приложении, не беспокоясь о коммуникационных протоколах, сбалансированности загрузки и других неприятностях программирования со статическими потоками. Параллельная платформа содержит планировщик, который автоматически обеспечивает баланс загрузки, тем самым существенно упрощая работу программиста. Хотя функциональность сред с динамической многопоточностью продолжает развиваться, почти все они поддерживают две возможности: вложенный параллелизм и параллельные циклы. Вложенный параллелизм обеспечивает параллельный запуск подпрограмм, позволяя вызывающей программе продолжать работу, пока запущенная подпрограмма выполняет свои вычисления. Параллельный цикл подобен обычному циклу **for**, с тем отличием, что его итерации могут выполняться одновременно.

Эти две возможности образуют базис модели динамической многопоточности, которую мы изучим в данной главе. Ключевым моментом этой модели является то, что программист должен указывать только логическую параллельность вычислений, потоки параллельной платформы и распределение вычислений между ними. Мы будем изучать многопоточные алгоритмы, написанные для этой модели, и то, каким образом параллельная платформа может эффективно планировать выполняемые вычисления.

Наша модель динамической многопоточности имеет несколько важных преимуществ.

- Она является простым расширением нашей последовательной модели. Мы можем описать многопоточный алгоритм, добавив в псевдокод только три ключевых слова: **parallel**, **spawn** и **sync**. Кроме того, если мы удалим эти слова из многопоточного псевдокода, то получим текст последовательного псевдокода для решения той же задачи, который мы будем называть сериализацией (serialization) многопоточного алгоритма.
- Она обеспечивает теоретически ясный способ количественного описания параллелизма на основе понятий работы (work) и интервала (span).
- Многие многопоточные алгоритмы включают вложенный параллелизм, естественным образом вытекающий из парадигмы "разделяй и властвуй". Кроме того, многопоточные алгоритмы, как и последовательные, могут быть проанализированы путем решения рекуррентных соотношений.
- Модель соответствует развитию практики параллельных вычислений. Все большее количество параллельных платформ поддерживают тот или иной вариант динамической многопоточности, включая Cilk [50, 117], Cilk++ [70], OpenMP [58], Task Parallel Library [229] и Threading Building Blocks [290].

В разделе 27.1 введена модель динамической многопоточности и представлены метрики работы, интервала и параллелизма, которые будут использоваться нами для анализа многопоточных алгоритмов. В разделе 27.2 изучается многопоточное умножение матриц, а раздел 27.3 посвящен сложной задаче многопоточной сортировки слиянием.

27.1. Основы динамической многопоточности

Начнем изучение динамической многопоточности с примера рекурсивного вычисления чисел Фибоначчи. Вспомним, что числа Фибоначчи определены рекуррентным соотношением (3.22):

$$F_0 = 0 ,$$
$$F_1 = 1 ,$$
$$F_i = F_{i-1} + F_{i-2} \qquad \text{для } i \geq 2 .$$

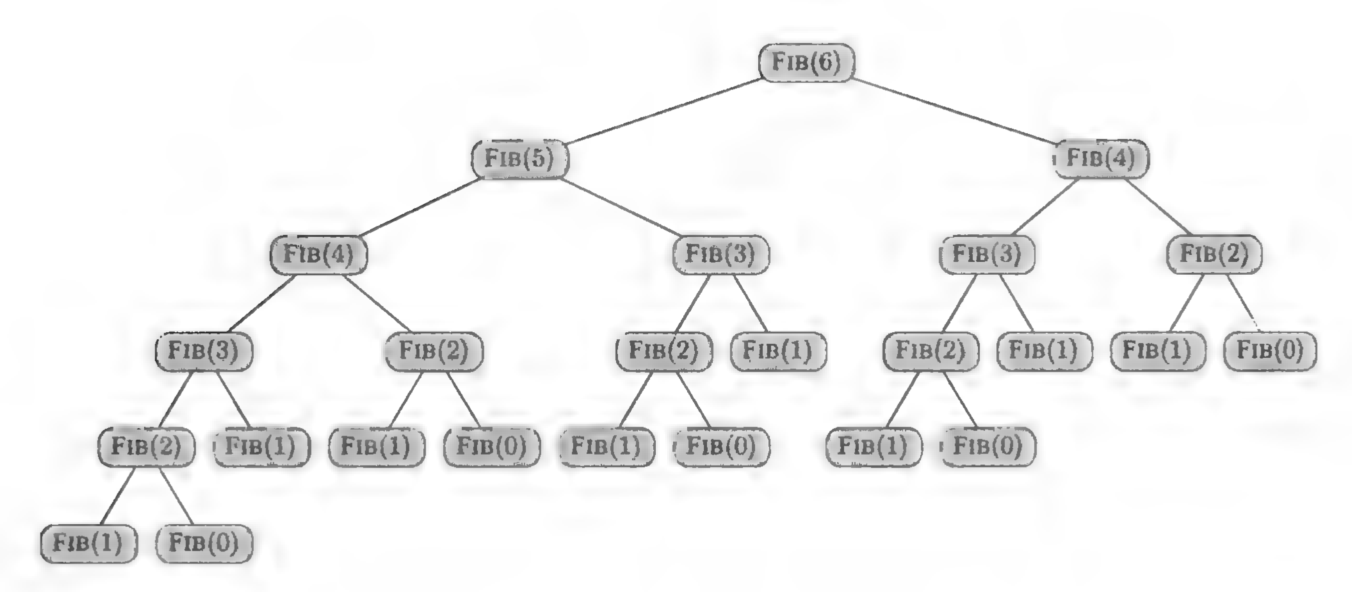


Рис. 27.1. Дерево экземпляров рекурсивной процедуры при вычислении FIB(6). Каждый экземпляр FIB с одинаковым аргументом выполняет одинаковую работу и возвращает одинаковый результат, так что этот способ вычисления чисел Фибоначчи очень неэффективен, но поучителен.

Вот простой, рекурсивный последовательный алгоритм вычисления n-го числа Фибоначчи.

```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
    y = FIB(n − 2)
    return x + y
```

На практике вы не должны вычислять числа Фибоначчи таким способом, поскольку при таком вычислении имеется очень много повторяющейся работы. На рис. 27.1 показано дерево экземпляров рекурсивной процедуры, создаваемых при вычислении F_6. Например, вызов FIB(6) рекурсивно вызывает сначала FIB(5), а затем — FIB(4). Но вызов FIB(5) также вызывает FIB(4). Оба экземпляра FIB(4) возвращают один и тот же результат ($F_4 = 3$). Поскольку процедура FIB не использует технологию запоминания (memoization), второй вызов FIB(4) вновь выполняет всю работу, уже выполненную первым вызовом.

Обозначим через $T(n)$ время работы процедуры FIB(n). Поскольку FIB(n) содержит два рекурсивных вызова плюс константное количество дополнительной работы, мы получим рекуррентное соотношение

$$T(n) = T(n-1) + T(n-2) + \Theta(1) .$$

Это рекуррентное соотношение имеет решение $T(n) = \Theta(F_n)$, что можно показать с помощью метода подстановки. В качестве гипотезы индукции примем, что $T(n) \leq aF_n - b$, где $a > 1$ и $b > 0$ представляют собой константы. При

подстановке получим

$$\begin{aligned} T(n) &\leq (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &= aF_n - b - (b - \Theta(1)) \\ &\leq aF_n - b , \end{aligned}$$

если выберем b достаточно большим для доминирования над константой в $\Theta(1)$. Затем мы можем выбрать a достаточно большим для удовлетворения начальному условию. Аналитическая граница

$$T(n) = \Theta(\phi^n) , \tag{27.1}$$

где $\phi = (1 + \sqrt{5})/2$ представляет собой золотое сечение, теперь следует из уравнения (3.25). Поскольку F_n растет с ростом n экспоненциально, эта процедура представляет собой очень медленный способ вычисления чисел Фибоначчи. (См. гораздо более быстрые способы в задаче 31.3.)

Хотя процедура FIB — не лучший, если не худший способ вычисления чисел Фибоначчи, она представляет собой хороший пример для иллюстрации ключевых концепций анализа многопоточных алгоритмов. Заметим, что в FIB(n) два рекурсивных вызова, FIB$(n-1)$ и FIB$(n-2)$, в строках 3 и 4 являются независимыми один от другого: они могут быть вызваны в любом порядке, и вычисления одной процедуры никак не влияют на вычисления другой. Таким образом, эти два рекурсивных вызова могут работать параллельно.

Расширим используемый в книге псевдокод, добавив новые ключевые слова, а именно — **spawn** и **sync**. Вот как можно переписать процедуру FIB с использованием динамической многопоточности.

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4      y = P-FIB(n − 2)
5      sync
6      return x + y
```

Обратите внимание, что если мы уберем ключевые слова **spawn** и **sync** из псевдокода P-FIB, то полученный в результате псевдокод будет идентичен FIB (не считая другого имени процедуры и двух рекурсивных вызовов). Определим ***сериализацию*** (serialization) многопоточного алгоритма как последовательный алгоритм, получаемый при удалении многопоточных ключевых слов **spawn** и **sync** (а когда мы изучим параллельные циклы, еще и **parallel**). Фактически наш псевдокод обладает тем приятным свойством, что сериализация всегда представляет собой обычный последовательный код, решающий ту же самую задачу.

Вложенный параллелизм осуществляется, когда ключевое слово **spawn** предшествует вызову процедуры, как в строке 3. Семантика запуска (spawn) отличает-

ся от обычного вызова процедуры тем, что экземпляр процедуры, выполняющий запуск (***родительская*** процедура), может продолжать работать параллельно с запущенной ***дочерней*** подпрограммой, вместо того чтобы ожидать завершения ее работы, как это обычно делается при последовательном выполнении. В нашем случае, пока запущенная дочерняя подпрограмма вычисляет P-FIB$(n-1)$, родительская процедура может продолжать вычисление P-FIB$(n-2)$ в строке 4 параллельно с запущенной дочерней подпрограммой. Поскольку процедура P-FIB рекурсивна, эти два вызова сами создают вложенный параллелизм, как и их дочерние подпрограммы, тем самым создавая потенциально огромное дерево параллельно выполняемых подвычислений.

Однако ключевое слово **spawn** не требует от процедуры *обязательного* параллельного с запущенной дочерней подпрограммой выполнения; оно лишь *разрешает* таковое. Ключевые слова для параллельных вычислений выражают ***логический параллелизм*** (logical parallelism) вычислений, указывая, какие части вычислений могут выполняться параллельно. Во время выполнения программы ***планировщик*** (scheduler) определяет, какие подвычисления в действительности могут выполняться параллельно, назначая их доступным процессорам. Немного позже мы рассмотрим теоретические основы работы планировщиков.

Процедура не может безопасно использовать значения, возвращаемые запущенными дочерними подпрограммами, кроме как после выполнения команды **sync**, как в строке 5. Ключевое слово **sync** указывает, что для того, чтобы перейти к следующей за **sync** команде, процедура должна дождаться окончания всех запущенных дочерних подпрограмм. В процедуре P-FIB ключевое слово **sync** необходимо перед ключевым словом **return** в строке 6 для того, чтобы избежать некорректного суммирования x и y до того, как будет вычислено значение x. В дополнение к явной синхронизации, обеспечиваемой ключевым словом **sync**, каждая процедура выполняет **sync** перед возвратом неявно, таким образом гарантируя, что все дочерние подпрограммы будут завершены до окончания родительской процедуры.

Модель многопоточного выполнения

Облегчить понимание ***многопоточного вычисления*** (multithreaded computation), которое является ни чем иным, как множеством команд, выполняемых процессором от имени многопоточной программы, может его представление в виде ориентированного ациклического графа $G = (V, E)$, именуемого графом вычислений. В качестве примера на рис. 27.2 показан граф вычислений P-FIB(4). Концептуально вершинами в V являются команды, а ребра в E представляют зависимости между командами, где $(u, v) \in E$ означает, что команда u должна выполняться до команды v. Для удобства, однако, если цепочка команд не содержит параллельных управляющих инструкций (**spawn**, **sync** или **return**, где последняя инструкция представляет собой возврат из запущенной подпрограммы — как явный, с применением ключевого слова **return**, так и неявный возврат по достижении конца процедуры), их можно группировать в единый фрагмент (***strand***), каждый из которых представляет одну или несколько команд. Команды управления

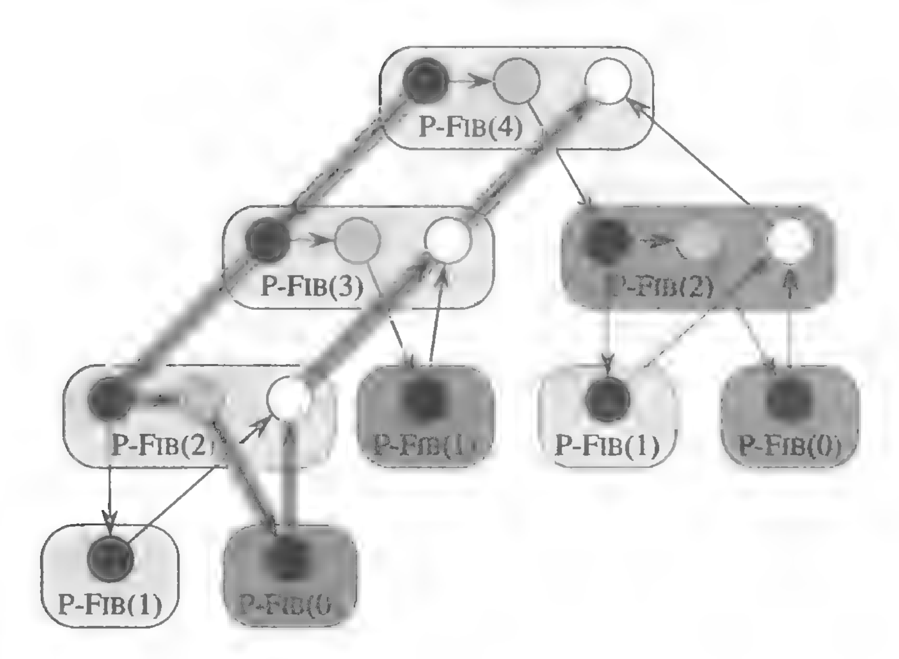


Рис. 27.2. Ориентированный ациклический граф, представляющий вычисление P-FIB(4). Каждый круг представляет один фрагмент вычисления, причем черные круги представляют либо базовый случай, либо часть процедуры (экземпляр) до запуска P-FIB($n-1$) в строке 3, серые круги представляют часть процедуры, которая вызывает P-FIB($n-2$) в строке 4 перед ключевым словом **sync** в строке 5, где процедура приостанавливается в ожидании возврата запущенной процедуры P-FIB($n-1$), а белые круги представляют часть процедуры после ключевого слова **sync**, в которой выполняется сложение значения x и y перед точкой, в которой происходит возврат полученного результата. Каждая группа фрагментов, принадлежащих одной и той же процедуре, ограничена скругленным прямоугольником, слабо заштрихованным для запускаемых процедур и сильно — для вызываемых. Ребра запуска и вызова направлены вниз, продолжения выполнения — по горизонтали вправо, а ребра возврата — вверх. Полагая, что каждый фрагмент требует единицы времени, вся работа составляет 17 единиц, поскольку имеется 17 фрагментов, а интервал равен 8 единицам, поскольку критический путь (указанный выделенными штриховкой ребрами) содержит 8 фрагментов.

параллельными вычислениями в фрагменты не входят, но представлены в структуре ориентированного ациклического графа. Например, если фрагмент имеет два преемника, один из них должен быть запущен параллельно, а для предшественника указывают, что эти предшественники объединяются инструкцией **sync**. Таким образом, в общем случае множество V образует множество фрагментов, а множество ориентированных ребер E представляет зависимости между фрагментами, порожденными управлением параллельными вычислениями. Если в G имеется ориентированный путь от фрагмента u к фрагменту v, мы говорим, что эти два фрагмента ***(логически) последовательны***. В противном случае фрагменты u и v ***(логически) параллельны***.

Многопоточное вычисление можно изобразить как ориентированный ациклический граф фрагментов, встроенный в дерево экземпляров процедур. Например, на рис. 27.1 показано дерево экземпляров процедур для вычисления P-FIB(6) (без детализированной структуры показанных фрагментов). На рис. 27.2 показана в увеличенном виде одна из частей этого дерева, с демонстрацией фрагментов, составляющих каждую процедуру. Все ориентированные ребра, соединяющие фрагменты, проходят либо в пределах процедуры, либо вдоль неориентированных ребер дерева процедуры.

Чтобы иметь возможность указывать различные зависимости между фрагментами, ребра ориентированного ациклического графа вычислений можно классифицировать. ***Ребро продолжения*** (continuation edge) (u, u'), на рис. 27.2 направленное горизонтально, соединяет фрагмент u с его преемником u' в пределах одного и того же экземпляра процедуры. Когда фрагмент u запускает фрагмент v, ориентированный ациклический граф содержит ***ребро запуска*** (spawn edge) (u, v), которое на рисунке направлено вниз. ***Ребра вызовов*** (call edges), представляя обычные вызовы процедур, также направлены вниз. Фрагмент u, запускающий фрагмент v, отличается от фрагмента u, вызывающего фрагмент v тем, что запуск приводит к наличию горизонтального ребра продолжения от фрагмента u к фрагменту u', следующему в процедуре за u и указывающему, что u' можно выполнять одновременно с v, в то время как вызов не приводит к появлению такого ребра. Когда фрагмент u выполняет возврат в вызвавшую процедуру, а x представляет собой фрагмент, непосредственно следующий за следующим **sync** в вызывающей процедуре, ориентированный ациклический граф вычислений содержит ***ребро возврата*** (return edge) (u, x), направленное вверх. Вычисление начинается с одного ***начального фрагмента*** (initial strand) — черной вершины в процедуре, помеченной как P-FIB(4) на рис. 27.2 — и заканчивается единственным *конечным фрагментом* (final strand) — белой вершиной в процедуре, помеченной как P-FIB(4).

Мы будем изучать выполнение многопоточных алгоритмов на ***идеальном параллельном компьютере*** (ideal parallel computer), состоящем из множества процессоров и ***последовательно согласованной*** (sequentially consistent) совместно используемой памяти. Последовательная согласованность означает, что совместно используемая память, в которую в действительности процессорами одновременно может выполняться много как загрузок, так и чтений, дает одни и те же результаты, как и в случае, если на каждом шаге будет выполняться только одна команда одного процессора. Иначе говоря, память ведет себя так, как если бы команды выполнялись последовательно в некотором глобальном линейном порядке, сохраняющем индивидуальный порядок выполнения команд каждым процессором. В случае динамических многопоточных вычислений, автоматически распределяемых по процессорам параллельной платформой, совместно используемая память ведет себя так, как если бы команды многопоточных вычислений чередовались таким образом, что образовывали бы линейный порядок, сохраняющий частичный порядок ориентированного ациклического графа вычислений. В зависимости от планировщика упорядочение может отличаться при разных запусках одной и той же программы, но поведение любого выполнения программы может быть понято в предположении, что команды выполняются в некотором линейном порядке, согласованном с ориентированным ациклическим графом вычислений.

В дополнение к сделанным семантическим предположениям модель идеального параллельного компьютера делает также некоторые предположения, связанные с производительностью. В частности, предполагается, что каждый процессор машины обладает одной и той же вычислительной мощностью, а стоимость планирования игнорируется. Хотя это последнее предположение может казаться слишком оптимистичным, оказывается, что для алгоритмов с достаточной степе-

нью "параллелизма" (термин, который вскоре будет точно определен) на практике накладные расходы планирования обычно минимальны.

Меры производительности

Измерять теоретическую эффективность многопоточного алгоритма можно с использованием двух метрик: "работы" (work) и "интервала" (span). ***Работа*** многопоточного вычисления представляет собой общее время выполнения всего вычисления на одном процессоре. Другими словами, работа представляет собой сумму времен, взятую по всем фрагментам. В ориентированном ациклическом графе вычислений, в котором каждый фрагмент требует единичного времени, работа равна просто количеству вершин ориентированного ациклического графа. ***Интервал*** представляет собой наибольшее время выполнения фрагментов вдоль произвольного пути в ориентированном ациклическом графе. Опять же, в случае ориентированного ациклического графа, в котором каждый фрагмент требует единичного времени, интервал равен количеству вершин на наидлиннейшем, или ***критическом, пути*** (critical path) в ориентированном ациклическом графе. (Вспомним из раздела 24.2, что критический путь в ориентированном ациклическом графе $G = (V, E)$ можно найти за время $\Theta(V + E)$.) Например, ориентированный ациклический граф вычислений на рис. 27.2 всего имеет 17 вершин, а в критическом пути у него 8 вершин, так что если каждый фрагмент требует единичного времени, то работа равна 17 единицам времени, а интервал — 8 единицам.

Фактическое время выполнения многопоточного вычисления зависит не только от его работы и интервала, но и от количества доступных процессоров и от того, как планировщик распределяет фрагменты по процессорам. Обозначая время выполнения многопоточного вычисления на P процессорах, мы будем использовать P в качестве нижнего индекса. Например, время выполнения алгоритма на P процессорах может быть обозначено как T_P. Работа представляет собой время выполнения алгоритма на одном процессоре, т.е. T_1. Интервал является временем работы при выполнении каждого фрагмента на своем собственном процессоре — другими словами, если у нас имеется неограниченное количество процессоров, — так что интервал мы обозначаем как T_∞.

Работа и интервал предоставляют нам нижнюю границу времени выполнения T_P многопоточного вычисления на P процессорах.

- За один шаг идеальный параллельный компьютер с P процессорами может выполнить не более P единиц работы; таким образом, за время T_P он может выполнить работу, не превышающую PT_P. Поскольку общая работа составляет T_1, имеем $PT_P \geq T_1$. Деление на P дает ***правило работы***:

$$T_P \geq T_1/P . \qquad (27.2)$$

- Идеальный параллельный компьютер с P процессорами не может работать быстрее машины с неограниченным количеством процессоров. Так что машина с неограниченным количеством процессоров может эмулировать P-процессорную машину, используя только P из своих процессоров. Таким образом,

можно сформулировать следующее ***правило интервалов***:

$$T_P \geq T_\infty \ . \tag{27.3}$$

Определим ***ускорение*** (speedup) вычислений на P процессорах как отношение T_1/T_P, говорящее о том, во сколько раз вычисления на P процессорах быстрее, чем на 1 процессоре. Согласно правилу работы имеем $T_P \geq T_1/P$, откуда вытекает, что $T_1/T_P \leq P$. Таким образом, ускорение на P процессорах не может превышать P. Когда ускорение линейно зависит от количества процессоров, т.е. когда $T_1/T_P = \Theta(P)$, вычисления демонстрируют ***линейное ускорение***, а когда $T_1/T_P = P$, мы имеем ***идеальное линейное ускорение***.

Отношение T_1/T_∞ работы к интервалу дает ***параллелизм*** (parallelism) многопоточного вычисления. Параллелизм можно рассматривать с трех точек зрения. Как отношение параллелизм определяет среднее количество работы, которая может быть выполнена параллельно на каждом шаге вдоль критического пути. Как верхняя граница параллелизм дает максимально возможное ускорение, которое может быть достигнуто с помощью произвольного количества процессоров. Наконец, что, вероятно, наиболее важно, параллелизм указывает предел возможности достичь идеального линейного ускорения. В частности, когда количество процессоров превышает уровень параллелизма, вычисления не могут достичь идеального линейного ускорения. Чтобы понять это утверждение, предположим, что $P > T_1/T_\infty$, и в этом случае из правила интервалов вытекает, что ускорение удовлетворяет условию $T_1/T_P \leq T_1/T_\infty < P$. Кроме того, если число процессоров P в идеальном параллельном компьютере существенно превышает параллелизм — т.е. если $P \gg T_1/T_\infty$, — то $T_1/T_P \ll P$, так что ускорение гораздо меньше, чем количество процессоров. Другими словами, чем больше процессоров мы используем сверх параллелизма, тем менее идеальным становится ускорение.

В качестве примера рассмотрим вычисление P-FIB(4) на рис. 27.2 и будем считать, что каждый фрагмент требует единичного времени. Поскольку работа $T_1 = 17$, а интервал $T_\infty = 8$, параллелизм равен $T_1/T_\infty = 17/8 = 2.125$. Следовательно, достигнуть более чем удвоенного ускорения невозможно, сколько бы процессоров не использовалось для проведения вычислений. Мы увидим, что в случае бóльших входных размеров P-FIB(n) демонстрирует существенную степень параллелизма.

Определим ***зазор параллельности*** (parallel slackness) многопоточного вычисления на идеальном параллельном компьютере с P процессорами как отношение $(T_1/T_\infty)/P = T_1/(PT_\infty)$, которое представляет собой множитель, на который параллелизм вычисления превосходит количество процессоров в машине. Таким образом, если зазор меньше 1, мы не можем надеяться достичь идеального линейного ускорения, поскольку $T_1/(PT_\infty) < 1$ и из правила интервалов вытекает, что ускорение на P процессорах удовлетворяет условию $T_1/T_P \leq T_1/T_\infty < P$. В действительности при уменьшении зазора от 1 до 0 ускорение вычисления отклоняется от идеального линейного ускорения все сильнее и сильнее. Однако, если зазор больше 1, ограничением является работа, приходящаяся на один про-

цессор. Как мы увидим, при росте зазора от 1 хороший планировщик может все больше и больше приближаться к идеальному линейному ускорению.

Планирование

Хорошая производительность зависит не только от минимизации работы и интервала. Должно также выполняться эффективное распределение фрагментов по процессорам параллельной машины. Наша модель многопоточного программирования не предоставляет возможности указать, какие фрагменты должны выполняться тем или иным процессором. Вместо этого в вопросе распределения вычислений по процессорам мы полагаемся на планировщик параллельной платформы. На практике планировщик распределяет фрагменты по статическим потокам, а операционная система распределяет потоки по процессорам, но этот дополнительный уровень косвенности не является необходимым для нашего понимания процесса планирования. Мы можем просто представить, что планировщик параллельной платформы распределяет фрагменты по процессорам непосредственно.

Многопоточный планировщик должен выполнять планирование, не зная заранее, когда фрагменты будут запущены или когда они завершатся — он должен работать в ***оперативном*** (on-line) режиме. Кроме того, хороший планировщик работает распределенно, с учетом баланса загруженности процессоров. Существование хороших оперативных распределенных планировщиков доказано, но их анализ представляет собой сложную задачу.

Поэтому для простоты мы будем рассматривать ***централизованный*** (centralized) планировщик, которому в любой момент времени известно глобальное состояние вычисления. В частности, мы проанализируем ***жадные планировщики***, назначающие процессорам на каждом временно́м шаге максимально возможное количество фрагментов. Если на очередном шаге к выполнению готовы как минимум P фрагментов, мы говорим, что этот шаг является ***полным***, и жадный планировщик назначает любые P готовых фрагментов процессорам. В противном случае для выполнения готово менее P фрагментов, и мы говорим, что шаг ***неполон***, а планировщик назначает каждый готовый фрагмент своему процессору.

Согласно правилу работы наилучшее время выполнения, на которое мы можем надеяться при наличии P процессоров, составляет $T_P = T_1/P$, а согласно правилу интервала лучшее, на что можно надеяться, — $T_P = T_\infty$. В приведенной далее теореме показано, что жадный планировщик доказанно хорош в том плане, что достигает в качестве верхней границы суммы этих двух нижних границ.

Теорема 27.1

На идеальном параллельном компьютере с P процессорами жадный планировщик выполняет многопоточное вычисление с работой T_1 и интервалом T_∞ за время

$$T_P \le T_1/P + T_\infty \; . \tag{27.4}$$

Доказательство. Начнем с рассмотрения полных шагов. На каждом полном шаге P процессоров вместе выполняют общее количество работы, равное P. Предположим для доказательства от противного, что количество полных шагов

строго больше, чем $\lfloor T_1/P \rfloor$. Тогда общее количество работы на полных шагах составляет не менее

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P \lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P && \text{(из уравнения (3.8))} \\ &> T_1 && \text{(из неравенства (3.9))} . \end{aligned}$$

Таким образом, мы получили противоречие, заключающееся в том, что P процессоров должны выполнить бóльшую работу, чем требует вычисление, и это позволяет заключить, что количество полных шагов не превышает $\lfloor T_1/P \rfloor$.

Рассмотрим теперь неполный шаг. Пусть G является ориентированным ациклическим графом, представляющим все вычисление в целом, и без потери общности предположим, что каждый фрагмент выполняется за единицу времени. (Можно заменить каждый более длинный фрагмент цепочкой фрагментов единичной длины.) Пусть G' представляет собой подграф G, который все еще должен быть выполнен по состоянию на начало неполного шага, и пусть G'' — подграф, который останется невыполненным после этого неполного шага. Наидлиннейший путь в ориентированном ациклическом графе с необходимостью начинается в вершине с нулевой входной степенью. Поскольку неполный шаг жадного планировщика выполняет все фрагменты с нулевой входящей степенью в G', длина наидлиннейшего пути в G'' должна быть на 1 меньше, чем длина наидлиннейшего пути в G'. Другими словами, неполный шаг уменьшает интервал невыполненного ориентированного ациклического графа на 1. Следовательно, количество неполных шагов не превышает T_∞.

Поскольку каждый шаг является либо полным, либо неполным, теорема доказана. ■

Приведенное далее следствие из теоремы 27.1 показывает, что жадный планировщик всегда хорошо работает.

Следствие 27.2

Время выполнения T_P любого многопоточного вычисления при планировании выполнения жадным планировщиком на идеальном параллельном компьютере с P процессорами отличается от оптимального не более чем в 2 раза.

Доказательство. Обозначим через T_P^* время выполнения, получаемое путем оптимального планирования на машине с P процессорами, и пусть T_1 и T_∞ обозначают соответственно работу и интервал вычисления. Поскольку правила работы и интервала — неравенства (27.2) и (27.3) — дают нам $T_P^* \geq \max(T_1/P, T_\infty)$, из теоремы 27.1 вытекает, что

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max(T_1/P, T_\infty) \\ &\leq 2T_P^* . \end{aligned}$$

■

Очередное следствие показывает, что фактически жадный планировщик с ростом зазора достигает близкого к идеальному линейного ускорения любого многопоточного вычисления.

Следствие 27.3

Пусть T_P обозначает время выполнения многопоточного вычисления, полученное жадным планировщиком на идеальном параллельном компьютере с P процессорами, и пусть T_1 и T_∞ представляют собой соответственно работу и интервал вычисления. Тогда, если $P \ll T_1/T_\infty$, имеем $T_P \approx T_1/P$, или, что эквивалентно, ускорение приблизительно равно P.

Доказательство. Если мы предположим, что $P \ll T_1/T_\infty$, то мы также имеем $T_\infty \ll T_1/P$, а следовательно, теорема 27.1 дает нам $T_P \le T_1/P + T_\infty \approx T_1/P$. Поскольку правило работы (27.2) гласит, что $T_P \ge T_1/P$, мы заключаем, что $T_P \approx T_1/P$, или, что эквивалентно, что ускорение равно $T_1/T_P \approx P$. ■

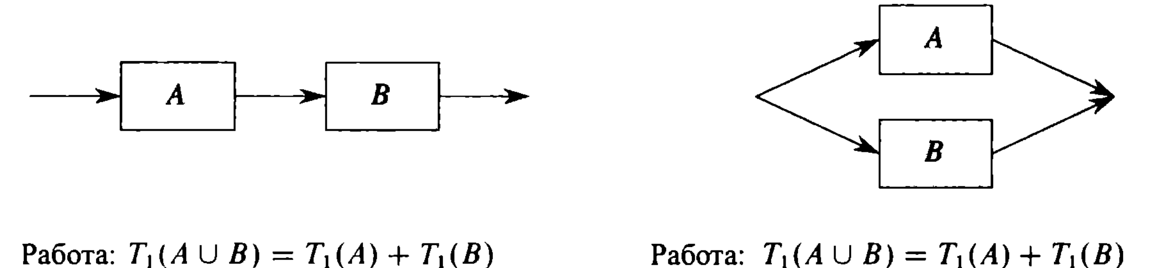
Символ $\ll$ означает "гораздо меньше", но насколько "гораздо"? На практике зазора, не меньшего 10 (т.е. когда параллелизм в 10 раз превышает количество процессоров), в общем случае достаточно для достижения хорошего ускорения. При этом член, соответствующий интервалу, в жадной границе (27.4) составляет менее 10% от члена, соответствующего работе, приходящейся на один процессор, что достаточно хорошо для большинства инженерных ситуаций. Например, если вычисление выполняется только на 10 или 100 процессорах, параллелизм, равный, скажем, 1 000 000, не имеет никакого преимущества над параллелизмом 10 000, несмотря на разницу в 100 раз. Как показано в задаче 27.2, иногда снижение чрезвычайно высокого параллелизма дает возможность получить алгоритмы, лучшие с других точек зрения, но при этом в той же степени масштабируемые на разумном количестве процессоров.

Анализ многопоточных алгоритмов

Теперь у нас есть все необходимые инструменты для проведения анализа многопоточных алгоритмов и получения точных границ времен их работы на различном количестве процессоров. Анализ работы достаточно прост и прямолинеен, поскольку ее подсчет — ни что иное, как анализ времени работы обычного последовательного алгоритма (а именно — сериализации многопоточного алгоритма), с которым вы уже должны быть хорошо знакомы, ведь именно этому вопросу посвящено большинство материала данной книги! Анализ интервала куда интереснее, но, вообще говоря, не сложнее — сто́ит только немного в нем разобраться. Основные идеи мы изучим на примере программы P-FIB.

Анализ работы $T_1(n)$ алгоритма P-FIB(n) не представляет трудности, поскольку мы уже его проводили. Исходная процедура FIB, по сути, является сериализацией P-FIB, а следовательно, $T_1(n) = T(n) = \Theta(\phi^n)$ согласно уравнению (27.1).

На рис. 27.3 показано, как выполняется анализ интервала. Если два подвычисления соединены последовательно, интервал их объединения равен сумме их интервалов; если же они соединены параллельно, то интервал их объединения равен



Работа: $T_1(A \cup B) = T_1(A) + T_1(B)$
Интервал: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(а)

Работа: $T_1(A \cup B) = T_1(A) + T_1(B)$
Интервал: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(б)

Рис. 27.3. Работа и интервал составных подвычислений. **(а)** Когда два подвычисления соединены последовательно, работа их объединения равна сумме отдельных работ, а интервал — сумме отдельных интервалов. **(б)** Когда два подвычисления соединены параллельно, работа их объединения равна сумме отдельных работ, а интервал — максимальному из отдельных интервалов.

максимальному из индивидуальных интервалов. В случае процедуры P-FIB(n) запускаемый вызов P-FIB$(n-1)$ в строке 3 выполняется параллельно с вызовом P-FIB$(n-2)$ в строке 4. Следовательно, интервал P-FIB(n) можно записать в виде рекуррентного соотношения

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) , \end{aligned}$$

решением которого является $T_\infty(n) = \Theta(n)$.

Параллелизм процедуры P-FIB(n) равен $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, т.е. быстро растет с ростом n. Таким образом, даже на очень больших параллельных компьютерах средних значений n достаточно для достижения почти идеального линейного ускорения для процедуры P-FIB(n), поскольку она обладает значительным зазором параллельности.

Параллельные циклы

Многие алгоритмы включают циклы, все итерации которых могут выполняться параллельно. Как мы увидим, такие циклы можно распараллелить с помощью ключевых слов **spawn** и **sync**, но гораздо удобнее явно указать, что итерации в таких циклах могут выполняться одновременно. В нашем псевдокоде такая функциональность обеспечивается ключевым словом **parallel**, предшествующим ключевому слову **for** в соответствующем цикле.

В качестве примера рассмотрим задачу умножения матрицы $A = (a_{ij})$ размером $n \times n$ на n-вектор $x = (x_j)$. Полученный в результате n-вектор $y = (y_i)$ определяется как

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

для $i = 1, 2, \ldots, n$. Вычислить произведение матрицы на вектор можно путем параллельного вычисления всех элементов y следующим образом.

```
MAT-VEC(A, x)
1  n = A.rows
2  y — вновь созданный вектор длиной n
3  parallel for i = 1 to n
4      y_i = 0
5  parallel for i = 1 to n
6      for j = 1 to n
7          y_i = y_i + a_ij x_j
8  return y
```

В этом коде ключевые слова **parallel for** в строках 3 и 5 указывают, что итерации соответствующих циклов могут выполняться одновременно. Компилятор может реализовать каждый цикл **parallel for** как подпрограмму с вложенным параллелизмом. Например, цикл **parallel for** в строках 5–7 может быть реализован с помощью вызова MAT-VEC-MAIN-LOOP$(A, x, y, n, 1, n)$ в стиле "разделяй и властвуй"; при этом компилятор генерирует вспомогательную подпрограмму MAT-VEC-MAIN-LOOP следующим образом.

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1  if i == i'
2      for j = 1 to n
3          y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋
5      spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6      MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
7      sync
```

Этот код рекурсивно запускает первую половину итераций цикла параллельно со второй половиной, а затем выполняет команду **sync**, тем самым создавая бинарное дерево выполнения, листья которого представляют отдельные итерации цикла, как показано на рис. 27.4.

Для вычисления работы $T_1(n)$ процедуры MAT-VEC в случае матрицы размером $n \times n$ мы просто вычисляем время работы сериализации этого алгоритма, получаемой простой заменой циклов **parallel for** обычными циклами **for**. Таким образом, мы имеем $T_1(n) = \Theta(n^2)$ из-за доминирующего квадратичного времени работы дважды вложенных циклов в строках 5–7. Этот анализ, однако, игнорирует накладные расходы, связанные с рекурсивным запуском в реализации с параллельными циклами. В действительности накладные расходы увеличивают работу параллельного цикла по сравнению с его сериализацией, но не асимптотически. Чтобы понять, почему это так, заметим, что поскольку дерево экземпляров рекурсивной процедуры является полным бинарным деревом, количество внутренних узлов на 1 меньше количества листьев (см. упр. Б.5.3). Каждый внутренний узел выполняет константную работу по разделению диапазона итераций, а каждый лист соответствует итерации цикла, которая требует как минимум константного времени ($\Theta(n)$ в данном случае). Таким образом, можно амортизировать наклад-

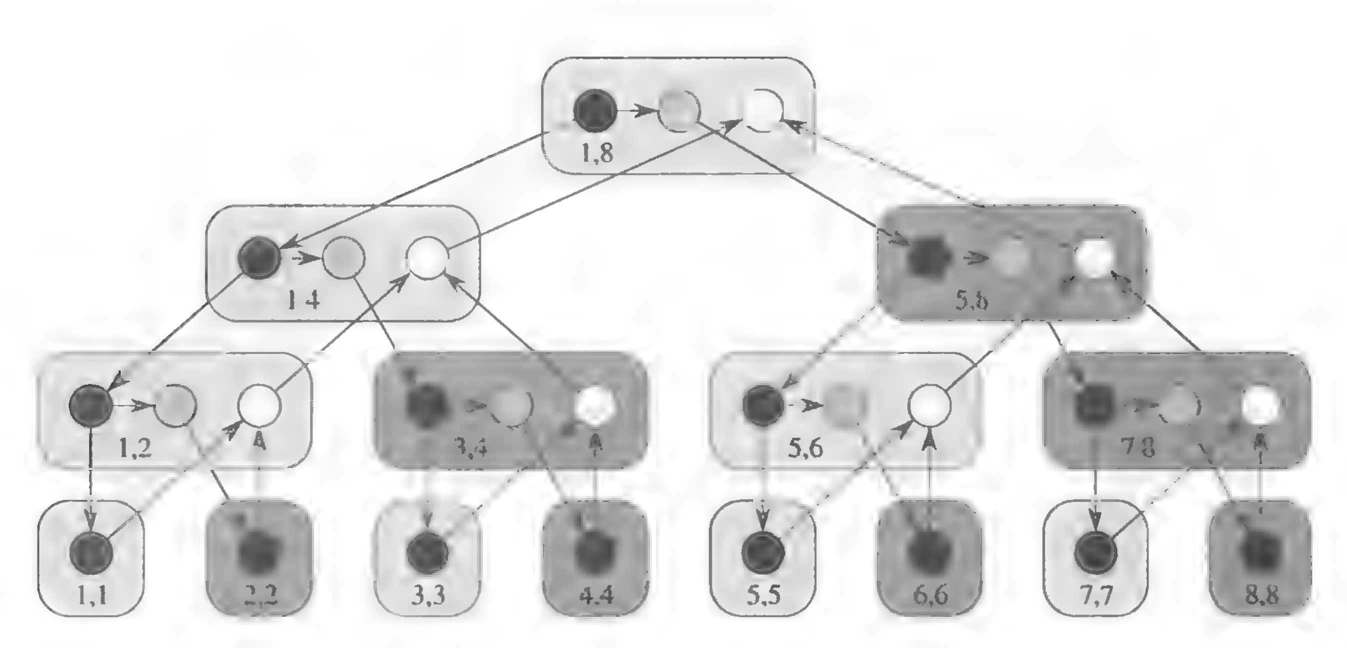


Рис. 27.4. Ориентированный ациклический граф, представляющий вычисление MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. Два числа внутри каждого скругленного прямоугольника указывают значения двух последних параметров (i и i' в заголовке процедуры) при запуске или вызове процедуры. Черные круги представляют фрагменты, соответствующие либо базовому случаю, либо части процедуры до запуска MAT-VEC-MAIN-LOOP в строке 5; серые круги представляют фрагменты, соответствующие части процедуры, которая вызывает MAT-VEC-MAIN-LOOP в строке 6 до ключевого слова **sync** в строке 7, где выполнение приостанавливается до тех пор, пока не будет осуществлен возврат подпрограммы, запущенной в строке 5; белые же круги представляют фрагменты, соответствующие (незначительной) части процедуры после ключевого слова **sync** и до точки возврата из процедуры.

ные расходы рекурсивного запуска, получая вклад, соответствующий не более чем добавлению постоянного множителя к общей работе.

С практической точки зрения динамически многопоточные параллельные платформы иногда ***огрубляют*** (coarsen) листья рекурсии, выполняя несколько итераций в пределах одного листа, либо автоматически, либо по команде программиста, тем самым снижая накладные расходы рекурсивного запуска. Правда, это снижение достигается ценой уменьшения уровня параллелизма, но если вычисление имеет достаточный зазор параллельности, близкое к идеальному линейное ускорение при этом в жертву не приносится.

Мы также должны учесть накладные расходы на рекурсивные запуски в ходе анализа интервала конструкции параллельного цикла. Поскольку глубина рекурсивных вызовов логарифмически зависит от количества итераций, в случае параллельного цикла с n итерациями, в котором i-я итерация имеет интервал $iter_\infty(i)$, интервал цикла равен

$$T_\infty(n) = \Theta(\lg n) + \max_{1 \le i \le n} iter_\infty(i) \ .$$

Например, для процедуры MAT-VEC в случае матрицы размером $n \times n$ параллельный цикл инициализации в строках 3 и 4 имеет интервал $\Theta(\lg n)$, поскольку рекурсивный запуск доминирует над константной работой в каждой итерации. Интервал дважды вложенных циклов в строках 5–7 составляет $\Theta(n)$, поскольку каждая итерация внешнего цикла **parallel for** содержит n итераций внутрен-

него (последовательного) цикла **for**. Интервал остального кода процедуры имеет константное значение, и, таким образом, доминирует интервал дважды вложенных циклов, что дает общий интервал всей процедуры $\Theta(n)$. Поскольку работа равна $\Theta(n^2)$, уровень параллелизма имеет значение $\Theta(n^2)/\Theta(n) = \Theta(n)$. (В упр. 27.1.6 предлагается построить реализацию с еще большим уровнем параллелизма.)

Условия гонки

Многопоточный алгоритм является ***детерминированным*** (deterministic), если он всегда приводит к одним и тем же результатам при одних и тех же входных данных, независимо от того, как именно планируются команды в многоядерном компьютере. Алгоритм является ***недетерминированным***, если его поведение может меняться от выполнения к выполнению. Зачастую многопоточные алгоритмы, которые должны быть детерминированными, таковыми не являются, поскольку содержат "гонку детерминированности" (determinacy race).

Гонка является проклятием параллельных вычислений. Среди знаменитых ошибок, связанных с ней, — прибор радиационной терапии Therac-25, убивший трех пациентов и серьезно навредивший еще нескольким, и североамериканское затмение 2003 года, когда без электричества остались более 50 миллионов человек. Эти пагубные ошибки очень трудно найти. Можно проводить различные многодневные тесты и так и не обнаружить спорадически проявляющиеся неполадки.

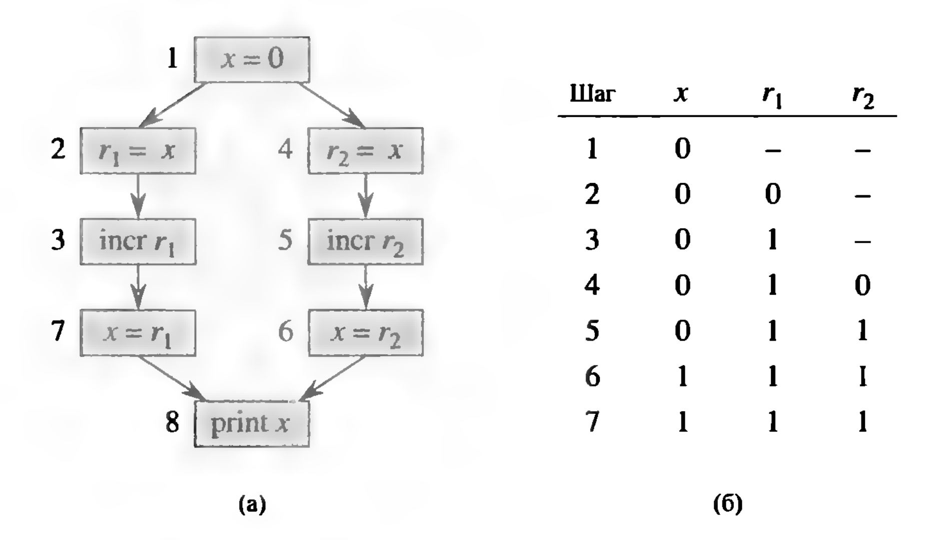
Гонка детерминированности осуществляется в том случае, когда две логически параллельные команды обращаются к одной и той же ячейке памяти и как минимум одна из них выполняет запись. Приведенная далее процедура иллюстрирует условие гонки.

```
RACE-EXAMPLE( )
1  x = 0
2  parallel for i = 1 to 2
3      x = x + 1
4  print x
```

После инициализации переменной x значением 0 в строке 1 процедура RACE-EXAMPLE создает два параллельных фрагмента, каждый из которых увеличивает x в строке 3. Хотя, на первый взгляд, процедура RACE-EXAMPLE должна всегда выводить значение 2 (ее сериализация так и делает), может оказаться выведенным и значение 1. Давайте разберемся, как это может случиться.

Когда процессор увеличивает значение переменной x, эта операция не является неделимой, а состоит из последовательности команд.

1. Чтение x из памяти в один из регистров процессора.
2. Увеличение значения в регистре.
3. Запись значения из регистра обратно в переменную x в памяти.



Шаг	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(б)

Рис. 27.5. Иллюстрация гонки детерминированности в процедуре RACE-EXAMPLE. **(а)** Ориентированный ациклический граф вычисления указывает зависимости между отдельными командами. r_1 и r_2 представляют собой регистры процессора. Команды, не относящиеся к гонке, такие как реализация управления циклом, опущены. **(б)** Последовательность выполнения, приводящая к ошибке, с указанием значения x в памяти и значений регистров r_1 и r_2 для каждого шага последовательности выполнения.

На рис. 27.5, (а) показан ориентированный ациклический граф вычисления, представляющий выполнение RACE-EXAMPLE, с фрагментами, разбитыми на отдельные команды. Вспомним, что, поскольку идеальный параллельный компьютер поддерживает последовательную согласованность, параллельное выполнение многопоточного алгоритма можно рассматривать как чередование команд с учетом зависимостей в ориентированном ациклическом графе. В части (б) рисунка показаны значения при выполнении вычисления, вызывающем аномалию. Значение x хранится в памяти, а r_1 и r_2 представляют собой регистры процессора. На первом шаге один из процессоров устанавливает значение x равным 0. На шагах 2 и 3 процессор 1 считывает значение x из памяти в свой регистр r_1 и увеличивает его, в результате чего в регистре r_1 хранится значение, равное 1. В этот момент в игру вступает процессор 2, выполняющий команды 4–6. Процессор 2 считывает значение x из памяти в свой регистр r_2 и увеличивает его, в результате чего в регистре r_2 хранится значение, равное 1; затем он сохраняет полученное значение в переменной x, присваивая ей значение 1. Теперь процессор 1 продолжает работу с шага 7, сохраняя значение 1 из регистра r_1 в переменную x, что оставляет значение x неизменным. Следовательно, на шаге 8 выводится значение 1, а не 2, как в случае сериализованной процедуры.

Мы видим, что получается. Если при параллельном выполнении процессор 1 выполняет свои команды до процессора 2, выводится значение 2. Если при параллельном выполнении, напротив, процессор 2 выполняет свои команды до процессора 1, выводится то же значение 2. Если же команды двух процессоров выполняются одновременно, то может случиться так, как в приведенном примере, когда одно из изменений значения x оказывается потерянным.

Конечно, множество выполнений не приводит к ошибке. Например, если порядок выполнения будет $\langle 1, 2, 3, 7, 4, 5, 6, 8 \rangle$ или $\langle 1, 4, 5, 6, 2, 3, 7, 8 \rangle$, мы получим корректный результат. В этом и заключается основная проблема гонки детерми-

нированности. В общем случае большинство порядков выполнения дают корректный результат — как в нашем случае, когда команды слева выполняются до команд справа или наоборот. Но некоторые порядки выполнения чередующихся команд приводят к неверным результатам. Из-за этого гонки крайне трудно тестировать. Можно выполнять тесты сутки напролет и ни разу не столкнуться с ошибкой, а затем получить катастрофические последствия при эксплуатации готового программного обеспечения.

Хотя справиться с проблемой гонки можно разными способами, включая применение взаимоисключающих блокировок и иных методов синхронизации, для наших целей мы просто будем гарантировать, что параллельно выполняемые фрагменты ***независимы***: между ними не может возникнуть гонка детерминированности. Таким образом, в конструкции **parallel for** все итерации должны быть независимыми. Между **spawn** и соответствующим ему **sync** запускаемый дочерний код должен быть независимым от кода родителя, включая код, выполняемый дополнительными запускаемыми или вызываемыми дочерними подпрограммами. Обратите внимание, что аргументы для запускаемой дочерней подпрограммы вычисляются родителем до того, как будет выполнен реальный запуск, так что вычисление этих аргументов происходит последовательно с любыми обращениями к ним после запуска.

В качестве примера того, насколько просто сгенерировать код с гонкой, приведем некорректную реализацию многопоточного умножения матрицы на вектор, интервал которой достигает значения $\Theta(\lg n)$ путем распараллеливания внутреннего цикла **for**.

```
MAT-VEC-WRONG(A, x)
1  n = A.rows
2  y — вновь созданный вектор длиной n
3  parallel for i = 1 to n
4      y_i = 0
5  parallel for i = 1 to n
6      parallel for j = 1 to n
7          y_i = y_i + a_ij x_j
8  return y
```

К сожалению, эта процедура некорректна из-за гонки при обновлении y_i в строке 7, выполняемой параллельно для всех n значений j. В упр. 27.1.6 предлагается разработать корректную реализацию алгоритма с интервалом $\Theta(\lg n)$.

Многопоточный алгоритм с гонкой иногда может оказаться корректным. Например, два параллельных потока могут сохранять одно и то же значение в совместно используемой переменной, и при этом не важно, какое значение будет сохранено первым. Однако в общем случае мы должны рассматривать код с гонкой как неверный.

Урок шахмат

Мы завершим этот раздел реальной историей, случившейся в процессе разработки многопоточной шахматной программы мирового уровня ⋆Socrates [79], хотя конкретные данные немного упрощены для ясности. Прототип программы отрабатывался на 32-процессорном компьютере, но окончательная работа предполагалась на суперкомпьютере с 512 процессорами. В некотором месте разработчики применили оптимизацию, которая позволила снизить время работы при тестировании на 32-процессорной машине с $T_{32} = 65$ до $T'_{32} = 40$ секунд. Затем разработчики использовали измерения работы и интервала и заключили, что эта оптимизированная версия, работающая быстрее на 32 процессорах, на самом деле окажется медленнее исходной версии при работе на машине с 512 процессорами. В результате такая "оптимизация" была отвергнута.

Вот как выглядел их анализ. Исходная версия программы имела работу $T_1 = 2048$ секунд и интервал $T_\infty = 1$ секунда. Если рассматривать неравенство (27.4) как равенство, $T_P = T_1/P + T_\infty$, и использовать его как приближение времени работы на P процессорах, то можно увидеть, что на самом деле $T_{32} = 2048/32 + 1 = 65$. При оптимизации работа становится равной $T'_1 = 1024$ секунды, а интервал — $T'_\infty = 8$ секунд. Вновь используя наше приближение, получаем $T'_{32} = 1024/32 + 8 = 40$.

Однако относительные скорости этих двух версий меняются, когда мы вычисляем время работы на 512 процессорах. В частности, мы имеем $T_{512} = 2048/512 + 1 = 5$ секунд, а $T'_{512} = 1024/512 + 8 = 10$ секунд. Оптимизация, которая ускоряет программу при работе на 32 процессорах, делает ее в два раза более медленной на 512 процессорах! Оптимизированная версия интервала равна 8, и недоминирующий член в формуле времени работы на 32 процессорах становится доминирующим в случае 512 процессоров, полностью сводя на нет преимущества использования большего количества процессоров.

Мораль этой истории в том, что работа и интервал могут предоставить лучшее средство экстраполяции производительности, чем время работы.

Упражнения

27.1.1

Предположим, что мы запускаем процедуру P-FIB$(n-2)$ в строке 4 процедуры P-FIB, а не вызываем ее, как это делается на самом деле. Как это повлияет на асимптотические работу, интервал и параллелизм?

27.1.2

Начертите ориентированный ациклический граф вычислений, получаемый при выполнении P-FIB(5). Считая, что каждый фрагмент вычисления выполняется за единичное время, вычислите работу, интервал и параллелизм данного вычисления. Покажите, как распределить полученный ориентированный ациклический граф по трем процессорам с использованием жадного планирования, пометив каждый фрагмент временны́м шагом, на котором он должен выполняться.

27.1.3

Докажите, что жадный планировщик обеспечивает следующую временну́ю границу, несколько более строгую, чем доказанная в теореме 27.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \ . \tag{27.5}$$

27.1.4

Постройте ориентированный ациклический граф вычислений, для которого одно выполнение жадным планировщиком может потребовать почти в два раза больше времени, чем другое выполнение жадным планировщиком на том же количестве процессоров. Опишите работу этих двух разных выполнений вычисления.

27.1.5

Профессор выполняет измерения своего детерминированного многопоточного алгоритма на идеальном параллельном компьютере с жадным планировщиком на 4, 10 и 64 процессорах. Он утверждает, что получил следующие величины: $T_4 = 80$ секунд, $T_{10} = 42$ секунды и $T_{64} = 10$ секунд. Докажите, что профессор либо врет, либо некомпетентен. (*Указание:* воспользуйтесь правилом работы (27.2), правилом интервала (27.3) и неравенством (27.5) из упр. 27.1.3.)

27.1.6

Разработайте многопоточный алгоритм для умножения матрицы размером $n \times n$ на n-вектор, который достигает уровня параллелизма $\Theta(n^2/\lg n)$ при работе $\Theta(n^2)$.

27.1.7

Рассмотрим следующий многопоточный псевдокод для транспонирования матрицы A размером $n \times n$ "на месте", без привлечения дополнительной памяти.

```
P-TRANSPOSE(A)
1  n = A.rows
2  parallel for j = 2 to n
3      parallel for i = 1 to j - 1
4          обменять a_ij с a_ji
```

Проанализируйте работу, интервал и уровень параллелизма этого алгоритма.

27.1.8

Предположим, что мы заменили цикл **parallel for** в строке 3 процедуры P-TRANSPOSE (см. упр. 27.1.7) обычным циклом **for**. Проанализируйте работу, интервал и уровень параллелизма полученного алгоритма.

27.1.9

Для какого количества процессоров две версии описанной выше шахматной программы будут иметь одинаковую скорость работы в предположении, что $T_P =$

27.2. Многопоточное умножение матриц

В этом разделе рассматривается многопоточное умножение матриц, сериализованное время работы которого изучалось в разделе 4.2. Мы изучим как многопоточные алгоритмы, основанные на стандартном тройном вложенном цикле, так и алгоритмы "разделяй и властвуй".

Многопоточное умножение матриц вложенными циклами

Первый алгоритм, который мы рассмотрим, — простейший алгоритм, основанный на распараллеливании циклов процедуры SQUARE-MATRIX-MULTIPLY, приведенной на с. 100.

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  C — вновь созданная матрица размером n × n
3  parallel for i = 1 to n
4      parallel for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Для того чтобы проанализировать этот алгоритм, заметим, что поскольку его сериализацией является алгоритм SQUARE-MATRIX-MULTIPLY, работа данного алгоритма представляет собой просто $T_1(n) = \Theta(n^3)$ — время работы алгоритма SQUARE-MATRIX-MULTIPLY. Интервал $T_\infty(n) = \Theta(n)$, поскольку алгоритм следует вниз по пути в дереве рекурсии для цикла **parallel for**, начинающегося в строке 3, затем вниз по дереву рекурсии для цикла **parallel for**, начинающегося в строке 4, после чего выполняет все n итераций обычного цикла **for**, начинающегося в строке 6, что в результате приводит к интервалу $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$. Таким образом, уровень параллелизма равен $\Theta(n^3)/\Theta(n) = \Theta(n^2)$. В упр. 27.2.3 предлагается распараллелить внутренний цикл для получения уровня параллелизма, равного $\Theta(n^3/\lg n)$, чего нельзя добиться простым использованием **parallel for**, поскольку при этом создаются гонки.

Многопоточный алгоритм "разделяй и властвуй" для умножения матриц

Как мы знаем из раздела 4.2, матрицы размером $n \times n$ можно перемножить последовательно за время $\Theta(n^{\lg 7}) = O(n^{2.81})$, воспользовавшись стратегией "разделяй и властвуй" Штрассена. Естественным образом возникает вопрос о многопоточной версии этого алгоритма. Начнем, как и в разделе 4.2, с многопоточной версии более простого алгоритма "разделяй и властвуй".

Вспомним (с. 102), что процедура SQUARE-MATRIX-MULTIPLY-RECURSIVE, которая умножает две матрицы, A и B, размером $n \times n$ и дает в результате матрицу C того же размера, основывается на разбиении каждой из трех матриц на

четыре подматрицы размером $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

После этого мы можем переписать умножение матриц следующим образом:

$$\begin{aligned} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}. \end{aligned} \tag{27.6}$$

Итак, для умножения двух матриц размером $n \times n$ мы выполняем восемь умножений матриц размером $n/2 \times n/2$ и одно сложение матриц размером $n \times n$. Приведенный ниже псевдокод реализует эту стратегию "разделяй и властвуй" с использованием вложенного параллелизма. В отличие от процедуры SQUARE-MATRIX-MULTIPLY-RECURSIVE, на которой она основана, процедура P-MATRIX-MULTIPLY-RECURSIVE получает выходную матрицу в качестве параметра, чтобы избежать излишнего выделения памяти для матриц.

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)

1 $n = A.rows$
2 **if** $n == 1$
3 $\quad c_{11} = a_{11}b_{11}$
4 **else** пусть T представляет собой новую матрицу размером $n \times n$
5 $\quad$ Разбиение A, B, C и T на подматрицы размером $n/2 \times n/2$
$\quad\quad A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; $C_{11}, C_{12}, C_{21}, C_{22}$;
$\quad\quad$ и $T_{11}, T_{12}, T_{21}, T_{22}$; соответственно
6 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(C_{11}, A_{11}, B_{11})
7 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(C_{12}, A_{11}, B_{12})
8 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(C_{21}, A_{21}, B_{11})
9 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(C_{22}, A_{21}, B_{12})
10 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(T_{11}, A_{12}, B_{21})
11 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(T_{12}, A_{12}, B_{22})
12 $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE(T_{21}, A_{22}, B_{21})
13 $\quad$ P-MATRIX-MULTIPLY-RECURSIVE(T_{22}, A_{22}, B_{22})
14 $\quad$ **sync**
15 $\quad$ **parallel for** $i = 1$ **to** n
16 $\quad\quad$ **parallel for** $j = 1$ **to** n
17 $\quad\quad\quad c_{ij} = c_{ij} + t_{ij}$

В строке 3 обрабатывается базовый случай умножения матриц размером 1×1. Рекурсивный случай обрабатывается в строках 4–17. Память для вре́менной матрицы T выделяется в строке 4, а в строке 5 выполняется разбиение каждой из матриц A, B, C и T на подматрицы размером $n/2 \times n/2$. (Как и в процедуре SQUARE-MATRIX-MULTIPLY-RECURSIVE на с. 102, мы обошли молчанием

27 Зак. 3726

небольшой вопрос о том, как использовать вычисления индексов для представления подматричных частей исходной матрицы.) Рекурсивный вызов в строке 6 присваивает подматрице C_{11} произведение подматриц $A_{11}B_{11}$, так что C_{11} становится равным первому из двух членов, образующих сумму в (27.6). Аналогично в строках 7–9 выполняется присваивание подматрицам C_{12}, C_{21} и C_{22} первого из двух членов соответствующих сумм в (27.6). В строке 10 подматрице T_{11} присваивается произведение подматриц $A_{12}B_{21}$, так что T_{11} становится равным второму члену в сумме для C_{11}. В строках 11–13 выполняется аналогичное присваивание вторых членов сумм для C_{12}, C_{21} и C_{22} подматрицам T_{12}, T_{21} и T_{22} соответственно. Первые семь рекурсивных вызовов запускаются, а последний выполняется в основном фрагменте. Инструкция **sync** в строке 14 гарантирует, что все произведения подматриц в строках 6–13 будут полностью вычислены, после чего мы добавляем произведения из T в C с использованием дважды вложенных циклов **parallel for** в строках 15–17.

Сначала проанализируем работу $M_1(n)$ процедуры P-MATRIX-MULTIPLY-RECURSIVE, проводя анализ времени работы ее последовательного предшественника SQUARE-MATRIX-MULTIPLY-RECURSIVE. В рекурсивном случае разбиение выполняется за время $\Theta(1)$, затем выполняется восемь рекурсивных умножений матриц размером $n/2 \times n/2$, и наконец выполняется работа $\Theta(n^2)$ по сложению двух матриц размером $n \times n$. Таким образом, рекуррентное соотношение для работы $M_1(n)$ имеет вид

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

согласно случаю 1 основной теоремы. Другими словами, работа нашего многопоточного алгоритма асимптотически та же, что и время выполнения процедуры SQUARE-MATRIX-MULTIPLY с трижды вложенными циклами из раздела 4.2.

Для определения интервала $M_\infty(n)$ процедуры P-MATRIX-MULTIPLY-RECURSIVE сначала заметим, что интервал для разбиения равен $\Theta(1)$ и над ним доминирует интервал $\Theta(\lg n)$ дважды вложенных циклов **parallel for** в строках 15–17. Поскольку восемь параллельных рекурсивных вызовов работают с матрицами одинакового размера, максимальным интервалом каждого рекурсивного вызова является интервал любого из них. Следовательно, рекуррентное соотношение для интервала $M_\infty(n)$ процедуры P-MATRIX-MULTIPLY-RECURSIVE имеет вид

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \ . \tag{27.7}$$

Это рекуррентное соотношение не подпадает ни под один из случаев основной теоремы, но соответствует условию упр. 4.6.2. Согласно упр. 4.6.2 решением рекуррентного соотношения (27.7) является $M_\infty(n) = \Theta(\lg^2 n)$.

Теперь, когда мы знаем работу и интервал процедуры P-MATRIX-MULTIPLY-RECURSIVE, можно вычислить уровень ее параллелизма как $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$ и увидеть, что он весьма высок.

Многопоточный метод Штрассена

В случае многопоточного алгоритма Штрассена мы следуем той же общей схеме, что и на с. 104, только с применением вложенного параллелизма.

1. Разбиваем входные матрицы A и B и выходную матрицу C на подматрицы размером $n/2 \times n/2$, как в (27.6). Работа и интервал этого шага составляют $\Theta(1)$ при использовании пересчета индексов.
2. Создаем 10 матриц $S_1, S_2, \ldots, S_{10}$, каждая из которых имеет размер $n/2 \times n/2$ и представляет собой сумму или разность двух матриц, созданных в п. 1. Все 10 матриц создаются ценой работы $\Theta(n^2)$ и интервала $\Theta(\lg n)$ с помощью дважды вложенных циклов **parallel for**.
3. Используя созданные в п. 1 подматрицы и 10 матриц, созданных в п. 2, рекурсивно запускаем вычисление семи произведений матриц $P_1, P_2, \ldots, P_7$ размером $n/2 \times n/2$.
4. Вычисляем искомые подматрицы $C_{11}, C_{12}, C_{21}, C_{22}$ результирующей матрицы C путем сложения и вычитания различных комбинаций матриц P_i, вновь используя дважды вложенные циклы **parallel for**. Вычисление всех четырех подматриц имеет работу $\Theta(n^2)$ и интервал $\Theta(\lg n)$.

Для анализа этого алгоритма сначала заметим, что, поскольку сериализация совпадает с исходным последовательным алгоритмом, работа нашего алгоритма равна времени работы сериализации, а именно — $\Theta(n^{\lg 7})$. Как и в случае процедуры P-MATRIX-MULTIPLY-RECURSIVE для интервала можно записать рекуррентное соотношение. В данном случае параллельно выполняются семь рекурсивных вызовов, но поскольку все они работают с матрицами одного и того же размера, мы получаем такое же рекуррентное соотношение (27.7), как и в случае процедуры P-MATRIX-MULTIPLY-RECURSIVE, решением которого является $\Theta(\lg^2 n)$. Таким образом, уровень параллелизма многопоточного алгоритма Штрассена равен $\Theta(n^{\lg 7}/\lg^2 n)$. Это высокий параллелизм, хотя и меньший, чем в случае процедуры P-MATRIX-MULTIPLY-RECURSIVE.

Упражнения

27.2.1

Изобразите ориентированный ациклический граф для вычислений для процедуры P-SQUARE-MATRIX-MULTIPLY для матриц размером 2×2, помечая соответствие вершин на своей диаграмме фрагментам выполнения алгоритма. Используйте соглашение, согласно которому ребра, соответствующие запускам и вызовам, направлены вниз, ребра продолжения направлены вправо, а ребра возвратов — вверх. Считая, что каждый фрагмент выполняется за единичное время, проанализируйте работу, интервал и параллелизм этого вычисления.

27.2.2

Повторите упр. 27.2.1 для процедуры P-MATRIX-MULTIPLY-RECURSIVE.

27.2.3

Запишите псевдокод многопоточного алгоритма, перемножающего две матрицы размером $n \times n$ с работой $\Theta(n^3)$, но с интервалом всего лишь $\Theta(\lg n)$. Проанализируйте свой алгоритм.

27.2.4

Запишите псевдокод эффективного многопоточного алгоритма, умножающего матрицу размером $p \times q$ на матрицу размером $q \times r$. Ваш алгоритм должен быть высокопараллельным, даже когда любые из значений p, q и r равны 1. Проанализируйте свой алгоритм.

27.2.5

Запишите псевдокод эффективного многопоточного алгоритма транспонирования матрицы размером $n \times n$ "на месте", без привлечения дополнительной памяти, с использованием для рекурсивного деления матрицы на четыре подматрицы размером $n/2 \times n/2$ метода "разделяй и властвуй" без цикла **parallel for**. Проанализируйте свой алгоритм.

27.2.6

Запишите псевдокод эффективной многопоточной реализации алгоритма (см. раздел 25.2), который вычисляет кратчайшие пути между всеми парами вершин в графе со взвешенными ребрами. Проанализируйте свой алгоритм.

27.3. Многопоточная сортировка слиянием

Мы познакомились с сортировкой слиянием в разделе 2.3.1, а в разделе 2.3.2 проанализировали время ее работы и показали, что оно составляет $\Theta(n \lg n)$. Поскольку сортировка слиянием сама по себе использует парадигму "разделяй и властвуй", складывается впечатление, что она является прекрасным кандидатом для многопоточности с применением вложенного параллелизма. Можно легко модифицировать псевдокод так, чтобы первый рекурсивный вызов был запускаемым.

```
MERGE-SORT'(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

Подобно своему последовательному аналогу, MERGE-SORT′ сортирует подмассив $A[p..r]$. После того как две рекурсивные подпрограммы в строках 3 и 4

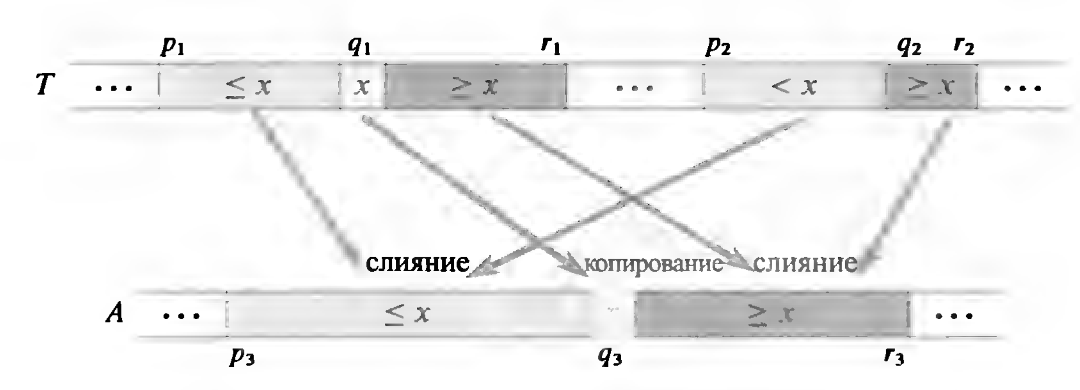


Рис. 27.6. Идея, лежащая в основе многопоточного слияния отсортированных подмассивов $T[p_1 \,..\, r_1]$ и $T[p_2 \,..\, r_2]$ в подмассив $A[p_3 \,..\, r_3]$. Пусть $x = T[q_1]$ представляет собой медиану $T[p_1 \,..\, r_1]$, а q_2 — место в $T[p_2 \,..\, r_2]$, такое, что x попадает между $T[q_2 - 1]$ и $T[q_2]$. Каждый элемент подмассивов $T[p_1 \,..\, q_1 - 1]$ и $T[p_2 \,..\, q_2 - 1]$ (светлая штриховка) не превышает x, а каждый элемент подмассивов $T[q_1 + 1 \,..\, r_1]$ и $T[q_2 + 1 \,..\, r_2]$ (темная штриховка) как минимум равен x. Для слияния мы вычисляем индекс q_3, где x располагается в $A[p_3 \,..\, r_3]$, копируем x в $A[q_3]$, а затем рекурсивно выполняем слияния $T[p_1 \,..\, q_1 - 1]$ с $T[p_2 \,..\, q_2 - 1]$ в $A[p_3 \,..\, q_3 - 1]$ и $T[q_1 + 1 \,..\, r_1]$ с $T[q_2 \,..\, r_2]$ в $A[q_3 + 1 \,..\, r_3]$.

будут завершены, что обеспечивается инструкцией **sync** в строке 5, процедура MERGE-SORT′ вызывает такую же процедуру MERGE, как и приведенная на с. 54.

Давайте проанализируем процедуру MERGE-SORT′. Для этого сначала проанализируем процедуру MERGE. Вспомним, что в последовательном случае время ее работы по слиянию n элементов составляет $\Theta(n)$. Поскольку процедура MERGE последовательная, и ее работа, и ее интервал равны $\Theta(n)$. Таким образом, работа $MS'_1(n)$ процедуры MERGE-SORT′ с n элементами характеризуется рекуррентным соотношением

$$\begin{aligned} MS'_1(n) &= 2\,MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

и совпадает со временем работы последовательной сортировки слиянием. Поскольку два рекурсивных вызова MERGE-SORT′ могут работать параллельно, интервал MS'_∞ определяется рекуррентным соотношением

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) \,. \end{aligned}$$

Таким образом, уровень параллелизма процедуры MERGE-SORT′ стремится к $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$, весьма невыразительной величине. Например, чтобы отсортировать 10 миллионов элементов, линейного ускорения можно достичь на нескольких процессорах, но эффективно использовать сотни процессоров не удастся.

Вероятно, вы уже догадались, в чем слабое место этой многопоточной реализации сортировки слиянием: в последовательной процедуре MERGE. Хотя слияние изначально может казаться по своей сути последовательным, в действительности можно создать его многопоточную версию с использованием вложенного параллелизма.

Наша стратегия "разделяй и властвуй" для многопоточного слияния, проиллюстрированная на рис. 27.6, работает с подмассивами массива T. Предположим, что мы сливаем два отсортированных подмассива — $T[p_1 \,..\, r_1]$ длиной $n_1 = r_1 - p_1 + 1$ и $T[p_2 \,..\, r_2]$ длиной $n_2 = r_2 - p_2 + 1$ — в подмассив $A[p_3 \,..\, r_3]$ длиной $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. Без потери общности мы делаем упрощающее предположение о том, что $n_1 \geq n_2$.

Сначала находим средний элемент $x = T[q_1]$ подмассива $T[p_1 \,..\, r_1]$, где $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Поскольку подмассив отсортирован, x является медианой $T[p_1 \,..\, r_1]$: любой элемент в $T[p_1 \,..\, q_1 - 1]$ не превышает x, а любой элемент в $T[q_1 + 1 \,..\, r_1]$ не меньше x. Затем мы используем бинарный поиск для того, чтобы найти индекс q_2 в подмассиве $T[p_2 \,..\, r_2]$, такой, чтобы подмассив оставался отсортированным, если мы вставим x между $T[q_2 - 1]$ и $T[q_2]$.

Затем мы сливаем исходные подмассивы $T[p_1 \,..\, r_1]$ и $T[p_2 \,..\, r_2]$ в $A[p_3 \,..\, r_3]$ следующим образом.

1. Устанавливаем $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
2. Копируем x в $A[q_3]$.
3. Рекурсивно сливаем $T[p_1 \,..\, q_1 - 1]$ с $T[p_2 \,..\, q_2 - 1]$ и размещаем результат в подмассиве $A[p_3 \,..\, q_3 - 1]$.
4. Рекурсивно сливаем $T[q_1 + 1 \,..\, r_1]$ с $T[q_2 \,..\, r_2]$ и размещаем результат в подмассиве $A[q_3 + 1 \,..\, r_3]$.

Когда мы вычисляем q_3, величина $q_1 - p_1$ представляет собой количество элементов в подмассиве $T[p_1 \,..\, q_1 - 1]$, а величина $q_2 - p_2$ — количество элементов в подмассиве $T[p_2 \,..\, q_2 - 1]$. Таким образом, их сумма равна количеству элементов, которые располагаются перед x в подмассиве $A[p_3 \,..\, r_3]$.

Мы имеем дело с базовым случаем, когда $n_1 = n_2 = 0$; при этом мы не должны выполнять никакой работы по слиянию двух пустых подмассивов. Поскольку мы предполагаем, что подмассив $T[p_1 \,..\, r_1]$ имеет как минимум ту же длину, что и $T[p_2 \,..\, r_2]$, т.е. $n_1 \geq n_2$, проверить базовый случай можно одной лишь проверкой $n_1 = 0$. Мы должны также убедиться, что рекурсия корректно обрабатывает случай, когда пуст только один из двух подмассивов, которым, в соответствии с нашим предположением о том, что $n_1 \geq n_2$, должен быть подмассив $T[p_2 \,..\, r_2]$.

Теперь превратим указанные идеи в псевдокод. Начнем с бинарного поиска, который выразим последовательно. Процедура BINARY-SEARCH(x, T, p, r) получает ключ x и подмассив $T[p \,..\, r]$ и возвращает одно из следующего списка.

- Если $T[p \,..\, r]$ пуст ($r < p$), то процедура возвращает индекс p.
- Если $x \leq T[p]$, а следовательно, не превышает все элементы $T[p \,..\, r]$, то процедура возвращает индекс p.
- Если $x > T[p]$, то процедура возвращает наибольший индекс q в диапазоне $p < q \leq r + 1$, такой, что $T[q - 1] < x$.

А вот и сам псевдокод.

```
BINARY-SEARCH(x, T, p, r)
low = p
high = max(p, r + 1)
while low < high
    mid = ⌊(low + high)/2⌋
    if x ≤ T[mid]
        high = mid
    else low = mid + 1
return high
```

Вызов BINARY-SEARCH(x, T, p, r) требует $\Theta(\lg n)$ последовательного времени в худшем случае (здесь $n = r - p + 1$ — размер подмассива, с которым работает процедура). (См. упр. 2.3.5.) Поскольку BINARY-SEARCH является последовательной процедурой, в наихудшем случае ее работа и интервал равны $\Theta(\lg n)$.

Теперь мы готовы написать псевдокод процедуры многопоточного слияния. Подобно процедуре MERGE на с. 54, в процедуре P-MERGE предполагается, что два сливаемых подмассива находятся в одном и том же массиве. Однако в отличие от процедуры MERGE, в процедуре P-MERGE не предполагается, что сливаемые подмассивы граничат друг с другом, т.е. процедура P-MERGE не требует выполнения равенства $p_2 = r_1 + 1$. Еще одно различие между MERGE и P-MERGE в том, что процедура P-MERGE получает в качестве аргумента выходной подмассив A, в котором должны будут храниться слитые значения. Вызов P-MERGE$(T, p_1, r_1, p_2, r_2, A, p_3)$ сливает два отсортированных подмассива, $T[p_1 \mathinner{..} r_1]$ и $T[p_2 \mathinner{..} r_2]$, в подмассив $A[p_3 \mathinner{..} r_3]$, где $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$ и не передается в качестве входного аргумента.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
1  n1 = r1 − p1 + 1
2  n2 = r2 − p2 + 1
3  if n1 < n2                    // Гарантируем, что n1 ≥ n2
4      Обменять p1 с p2
5      Обменять r1 с r2
6      Обменять n1 с n2
7  if n1 == 0                    // Оба пусты?
8      return
9  else q1 = ⌊(p1 + r1)/2⌋
10     q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11     q3 = p3 + (q1 − p1) + (q2 − p2)
12     A[q3] = T[q1]
13     spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14     P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15     sync
```

Процедура P-MERGE работает следующим образом. В строках 1 и 2 вычисляются длины n_1 и n_2 подмассивов $T[p_1 \mathinner{..} r_1]$ и $T[p_2 \mathinner{..} r_2]$ соответственно.

В строках 3–6 обеспечивается выполнение предположения о том, что $n_1 \geq n_2$. В строке 7 выполняется проверка базового случая, когда подмассив $T[p_1 \mathinner{\ldotp\ldotp} r_1]$ пуст (а следовательно, пуст и подмассив $T[p_2 \mathinner{\ldotp\ldotp} r_2]$), и в этом случае выполняется простой возврат из процедуры. В строках 9–15 реализована стратегия "разделяй и властвуй". Строка 9 вычисляет среднюю точку подмассива $T[p_1 \mathinner{\ldotp\ldotp} r_1]$, а строка 10 находит точку q_2 в подмассиве $T[p_2 \mathinner{\ldotp\ldotp} r_2]$, такую, что все элементы в $T[p_2 \mathinner{\ldotp\ldotp} q_2 - 1]$ меньше $T[q_1]$ (соответствующего x), а все элементы в $T[q_2 \mathinner{\ldotp\ldotp} r_2]$ не менее $T[q_1]$. В строке 11 вычисляется индекс q_3 элемента, который делит выходной подмассив $A[p_3 \mathinner{\ldotp\ldotp} r_3]$ на $A[p_3 \mathinner{\ldotp\ldotp} q_3 - 1]$ и $A[q_3 + 1 \mathinner{\ldotp\ldotp} r_3]$, а затем строка 12 копирует $T[q_1]$ непосредственно в $A[q_3]$.

Затем выполняется рекурсия с применением вложенного параллелизма. Строка 13 запускает первую подзадачу, в то время как строка 14 параллельно вызывает вторую подзадачу. Инструкция **sync** в строке 15 гарантирует, что обе подзадачи будут завершены до возврата из процедуры. (Поскольку каждая процедура неявно выполняет **sync** перед возвратом, можно опустить инструкцию **sync** в строке 15, но явное включение этой инструкции является хорошим стилем кодирования.) Имеется определенная тонкость при кодировании, обеспечивающая корректность работы при пустом подмассиве $T[p_2 \mathinner{\ldotp\ldotp} r_2]$. Она заключается в том, что на каждом рекурсивном вызове медианный элемент подмассива $T[p_1 \mathinner{\ldotp\ldotp} r_1]$ помещается в выходной подмассив, пока сам $T[p_1 \mathinner{\ldotp\ldotp} r_1]$ не станет, наконец, пустым и тем самым осуществится базовый случай.

Анализ многопоточного слияния

Сначала выведем рекуррентное соотношение для интервала $PM_\infty(n)$ процедуры P-MERGE, где два подмассива содержат в сумме $n = n_1 + n_2$ элементов. Поскольку запуск в строке 13 и вызов в строке 14 работают логически параллельно, рассмотрим только более дорогостоящий из них. Ключевым моментом является понимание того, что в наихудшем случае максимальное число элементов любого из рекурсивных вызовов может быть не более $3n/4$, что поясняется следующим образом. Поскольку строки 3–6 гарантируют, что $n_2 \leq n_1$, отсюда вытекает, что $n_2 = 2n_2/2 \leq (n_1 + n_2)/2 = n/2$. В наихудшем случае один из двух рекурсивных вызовов сливает $\lfloor n_1/2 \rfloor$ элементов подмассива $T[p_1 \mathinner{\ldotp\ldotp} r_1]$ со всеми n_2 элементами подмассива $T[p_2 \mathinner{\ldotp\ldotp} r_2]$, а следовательно, количество элементов, вовлеченных в вызов, составляет

$$\begin{aligned}
\lfloor n_1/2 \rfloor + n_2 &\leq n_1/2 + n_2/2 + n_2/2 \\
&= (n_1 + n_2)/2 + n_2/2 \\
&\leq n/2 + n/4 \\
&= 3n/4 \; .
\end{aligned}$$

Добавив стоимость $\Theta(\lg n)$ вызова BINARY-SEARCH в строке 10, мы получим следующее рекуррентное соотношение для интервала в наихудшем случае:

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) \; . \qquad (27.8)$$

(В базовом случае интервал равен $\Theta(1)$, поскольку строки 1–8 выполняются за константное время.) Это рекуррентное соотношение не подпадает ни под один из случаев основной теоремы, но соответствует условию из упр. 4.6.2. Следовательно, решением рекуррентного соотношения (27.8) является $PM_\infty(n) = \Theta(\lg^2 n)$.

Теперь проанализируем работу $PM_1(n)$ процедуры P-MERGE с n элементами, которая оказывается равной $\Theta(n)$. Поскольку каждый из n элементов должен быть скопирован из массива T в массив A, мы имеем $PM_1(n) = \Omega(n)$. Таким образом, остается только показать, что $PM_1(n) = O(n)$.

Сначала мы выведем рекуррентное соотношение для работы в наихудшем случае. Бинарный поиск в строке 10 стоит в наихудшем случае $\Theta(\lg n)$, что доминирует над прочей работой вне рекурсивных вызовов. Что касается последних, то заметим, что хотя рекурсивные вызовы в строках 13 и 14 могут сливать различные количества элементов, вместе эти два рекурсивных вызова сливают не более n элементов (в действительности — $n - 1$ элемент, поскольку $T[q_1]$ не участвует ни в одном из рекурсивных вызовов). Кроме того, как мы видели в ходе анализа интервала, рекурсивный вызов работает не более чем с $3n/4$ элементами. Поэтому мы получаем рекуррентное соотношение

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n) , \qquad (27.9)$$

где α лежит в диапазоне $1/4 \le \alpha \le 3/4$, и следует отдавать себе отчет, что фактическое значение α может меняться для каждого уровня рекурсии.

Докажем, что рекуррентное соотношение (27.9) имеет решение $PM_1 = O(n)$, с помощью метода подстановки. Будем считать, что $PM_1(n) \le c_1 n - c_2 \lg n$ для некоторых положительных констант c_1 и c_2. Подстановка дает

$$\begin{aligned}
PM_1(n) &\le (c_1 \alpha n - c_2 \lg(\alpha n)) + (c_1(1-\alpha)n - c_2 \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg \alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2 \lg n ,
\end{aligned}$$

поскольку мы можем выбрать c_2 достаточно большим, таким, чтобы $c_2(\lg n + \lg(\alpha(1-\alpha)))$ доминировало над членом $\Theta(\lg n)$. Кроме того, можно выбрать c_1 достаточно большим для удовлетворения базовым условиям рекуррентного соотношения. Поскольку работа $PM_1(n)$ процедуры P-MERGE равна и $\Omega(n)$, и $O(n)$, мы имеем $PM_1(n) = \Theta(n)$.

Уровень параллелизма процедуры P-MERGE равен $PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n)$.

Многопоточная сортировка слиянием

Теперь, когда у нас есть неплохо параллелизованная многопоточная процедура слияния, ее можно использовать в многопоточной сортировке слиянием. Эта версия сортировки слиянием подобна рассматривавшейся ранее процедуре

MERGE-SORT′, но в отличие от нее получает в качестве аргумента выходной подмассив B, в котором будут храниться отсортированные результаты. В частности, вызов P-MERGE-SORT(A, p, r, B, s) сортирует элементы в $A[p\,..\,r]$ и сохраняет их в $B[s\,..\,s+r-p]$.

```
P-MERGE-SORT(A, p, r, B, s)
1  n = r - p + 1
2  if n == 1
3      B[s] = A[p]
4  else T[1..n] — вновь созданный массив
5      q = ⌊(p + r)/2⌋
6      q' = q - p + 1
7      spawn P-MERGE-SORT(A, p, q, T, 1)
8      P-MERGE-SORT(A, q + 1, r, T, q' + 1)
9      sync
10     P-MERGE(T, 1, q', q' + 1, n, B, s)
```

После вычисления в строке 1 числа n элементов во входном подмассиве $A[p\,..\,r]$ в строках 2 и 3 обрабатывается базовый случай, когда массив содержит только один элемент. В строках 4–6 настраиваются рекурсивный запуск (в строке 7) и вызов (в строке 8), выполняющиеся параллельно. В частности, в строке 4 выделяется память для временного массива T с n элементами для хранения результатов рекурсивной сортировки слиянием. В строке 5 вычисляется индекс q в подмассиве $A[p\,..\,r]$, разделяющий элементы на два подмассива — $A[p\,..\,q]$ и $A[q+1\,..\,r]$, — которые будут отсортированы рекурсивно, а в строке 6 вычисляется количество q' элементов в первом подмассиве $A[p\,..\,q]$, которое в строке 8 используется для определения начального индекса в T, указывающего, где будут храниться отсортированные элементы подмассива $A[q+1\,..\,r]$. После этого выполняются рекурсивные запуск и вызов, после чего инструкция **sync** в строке 9 обеспечивает ожидание процедурой завершения запущенной подпрограммы. Наконец строка 10 вызывает P-MERGE для слияния отсортированных подмассивов, находящихся в $T[1\,..\,q']$ и $T[q'+1\,..\,n]$, в выходной подмассив $B[s\,..\,s+r-p]$.

Анализ многопоточной сортировки слиянием

Начнем с анализа работы $PMS_1(n)$ процедуры P-MERGE-SORT, который значительно проще анализа работы процедуры P-MERGE. Действительно, работа определяется рекуррентным соотношением

$$\begin{aligned} PMS_1(n) &= 2\,PMS_1(n/2) + PM_1(n) \\ &= 2\,PMS_1(n/2) + \Theta(n)\ . \end{aligned}$$

Это рекуррентное соотношение такое же, как и рекуррентное соотношение (4.4) для обычной процедуры MERGE-SORT из раздела 2.3.1 и имеет решение $PMS_1(n) = \Theta(n \lg n)$ согласно случаю 2 основной теоремы.

Теперь выведем и проанализируем рекуррентное соотношение для интервала $PMS_{\infty}(n)$ в наихудшем случае. Поскольку два рекурсивных вызова P-MERGE-SORT в строках 7 и 8 работают логически параллельно, можно игнорировать один из них, получив, таким образом, рекуррентное соотношение

$$\begin{aligned} PMS_{\infty}(n) &= PMS_{\infty}(n/2) + PM_{\infty}(n) \\ &= PMS_{\infty}(n/2) + \Theta(\lg^2 n) . \end{aligned} \qquad (27.10)$$

Как и в случае рекуррентного соотношения (27.8), основная теорема неприменима к рекуррентному соотношению (27.10), но мы можем воспользоваться упр. 4.6.2. Решение рекуррентного соотношения имеет вид $PMS_{\infty}(n) = \Theta(\lg^3 n)$, так что интервал процедуры P-MERGE-SORT равен $\Theta(\lg^3 n)$.

Параллельное слияние обеспечивает процедуру P-MERGE-SORT значительным преимуществом в плане параллелизма по сравнению с процедурой MERGE-SORT′. Вспомним, что уровень параллелизма процедуры MERGE-SORT′, которая вызывает последовательную процедуру MERGE, составляет всего $\Theta(\lg n)$. В случае процедуры P-MERGE-SORT уровень параллелизма равен

$$\begin{aligned} PMS_1(n)/PMS_{\infty}(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) , \end{aligned}$$

что гораздо лучше как с теоретической, так и с практической точек зрения. Хорошая практическая реализация может пожертвовать некоторой долей параллелизма, огрубляя базовый случай, чтобы снизить скрытую в асимптотических обозначениях константу. Наиболее простой путь такого огрубления базового случая — при достаточно малом размере массива использовать обычную последовательную сортировку (вероятно, наиболее подходящий вариант — быстрая сортировка).

Упражнения

27.3.1

Поясните, как огрубить базовый случай процедуры P-MERGE.

27.3.2

Вместо поиска медианного элемента в большем подмассиве, как это делается в процедуре P-MERGE, рассмотрите вариацию алгоритма, в которой выполняется поиск медианного элемента двух отсортированных подмассивов с использованием результата упр. 9.3.8. Запишите псевдокод эффективной многопоточной процедуры слияния, которая использует такую видоизмененную процедуру поиска медианы. Проанализируйте свой алгоритм.

27.3.3

Разработайте эффективный многопоточный алгоритм для разбиения массива относительно опорного элемента, как это делается в процедуре PARTITION на с. 199. От вас не требуется разбиение массива "на месте", без привлечения дополнительной памяти. Сделайте свой алгоритм максимально параллелизуемым. Проанали-

зируйте его. (*Указание:* вам может понадобиться вспомогательный массив и выполнение более одного прохода по входным элементам.)

27.3.4

Разработайте многопоточную версию процедуры RECURSIVE-FFT, приведенной на с. 953. Сделайте свой алгоритм максимально параллелизуемым. Проанализируйте его.

27.3.5 ★

Разработайте многопоточную версию процедуры RANDOMIZED-SELECT, приведенной на с. 246. Сделайте свой алгоритм максимально параллелизуемым. Проанализируйте его. (*Указание:* воспользуйтесь алгоритмом разбиения из упр. 27.3.3.)

27.3.6 ★

Покажите, как сделать многопоточным алгоритм SELECT из раздела 9.3. Сделайте свой алгоритм максимально параллелизуемым. Проанализируйте его.

Задачи

27.1. Реализация параллельных циклов с использованием вложенного параллелизма

Рассмотрим следующий многопоточный алгоритм для выполнения попарного сложения n-элементных массивов $A[1\,..\,n]$ и $B[1\,..\,n]$ с сохранением суммы в $C[1\,..\,n]$.

```
SUM-ARRAYS(A, B, C)
parallel for i = 1 to A.length
    C[i] = A[i] + B[i]
```

a. Перепишите параллельный цикл в SUM-ARRAYS с использованием вложенного параллелизма (**spawn** и **sync**) в духе процедуры MAT-VEC-MAIN-LOOP. Проанализируйте уровень параллелизма своей реализации.

Рассмотрим следующую альтернативную реализацию параллельного цикла, которая содержит некоторое задаваемое значение *grain-size*.

```
SUM-ARRAYS'(A, B, C)
n = A.length
grain-size = ?                // Задается отдельно
r = ⌈n/grain-size⌉
for k = 0 to r − 1
    spawn ADD-SUBARRAY(A, B, C, k · grain-size + 1,
                        min((k + 1) · grain-size, n))
sync
```

```
ADD-SUBARRAY(A, B, C, i, j)
for k = i to j
    C[k] = A[k] + B[k]
```

б. Предположим, что мы устанавливаем *grain-size* $= 1$. Каков уровень параллелизма этой реализации?

в. Запишите формулу, выражающую интервал процедуры SUM-ARRAYS$'$ через n и *grain-size*. Получите значение *grain-size*, максимизирующее уровень параллелизма.

27.2. Экономия временно используемой памяти при умножении матриц

Процедура P-MATRIX-MULTIPLY-RECURSIVE имеет тот недостаток, что она должна выделять память для вре́менной матрицы T размером $n \times n$, что может отрицательно сказаться на значении константы, скрытой в Θ-обозначении. Однако процедура P-MATRIX-MULTIPLY-RECURSIVE обладает высоким уровнем параллелизма. Например, если игнорировать константы в Θ-обозначении, параллелизм при умножении матриц размером 1000×1000 достигает около $1000^3/10^2 = 10^7$, поскольку $\lg 1000 \approx 10$. Большинство параллельных компьютеров имеет существенно меньше 10 миллионов процессоров...

а. Разработайте рекурсивный многопоточный алгоритм, который обходится без вре́менной матрицы T ценой увеличения интервала до $\Theta(n)$. (*Указание:* вычислите $C = C + AB$, следуя общей стратегии процедуры P-MATRIX-MULTIPLY-RECURSIVE, но инициализируя C параллельно и вставляя **sync** в тщательно выбранные места.)

б. Запишите и решите рекуррентные соотношения для работы и интервала своей реализации.

в. Проанализируйте уровень параллелизма своей реализации. Оцените, игнорируя константы в Θ-обозначениях, уровень параллелизма для матриц размером 1000×1000. Сравните его с уровнем параллелизма процедуры P-MATRIX-MULTIPLY-RECURSIVE.

27.3. Многопоточные матричные алгоритмы

а. Разработайте параллельный вариант процедуры LU-DECOMPOSITION, приведенной на с. 861, и запишите псевдокод многопоточной версии этого алгоритма. Добейтесь максимального уровня параллелизма своей реализации и проанализируйте ее работу, интервал и уровень параллелизма.

б. Выполните то же самое для процедуры LUP-DECOMPOSITION, приведенной на с. 864.

в. Выполните то же самое для процедуры LUP-SOLVE, приведенной на с. 857.

г. Выполните то же самое для многопоточного алгоритма, основанного на уравнении (28.13) для обращения симметричной положительно определенной матрицы.

27.4. Многопоточные приведение и вычисление префикса

⊗-приведением (⊗-reduction) массива $x[1\,..\,n]$, где ⊗ представляет собой ассоциативный оператор, является значение

$$y = x[1] \otimes x[2] \otimes \cdots \otimes x[n] \; .$$

Приведенная далее процедура вычисляет ⊗-приведение подмассива $x[i\,..\,j]$ последовательно.

```
REDUCE(x, i, j)
y = x[i]
for k = i + 1 to j
    y = y ⊗ x[k]
return y
```

а. Воспользуйтесь вложенным параллелизмом для реализации многопоточного алгоритма P-REDUCE, который выполняет те же действия с работой $\Theta(n)$ и интервалом $\Theta(\lg n)$. Проанализируйте свой алгоритм.

Еще одна задача связана с ***вычислением ⊗-префикса*** (⊗-prefix computation), иногда именуемого ***⊗-сканированием*** (⊗-scan), для массива $x[1\,..\,n]$, где ⊗ представляет собой ассоциативный оператор. ⊗-сканирование генерирует массив $y[1\,..\,n]$, определяемый как

$$\begin{aligned} y[1] &= x[1] \; , \\ y[2] &= x[1] \otimes x[2] \; , \\ y[3] &= x[1] \otimes x[2] \otimes x[3] \; , \\ &\vdots \\ y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n] \; , \end{aligned}$$

т.е. все префиксы массива x "суммируются" с использованием оператора ⊗. Вычисление ⊗-префикса можно выполнить с помощью следующей последовательной процедуры SCAN.

```
SCAN(x)
n = x.length
y[1..n] — вновь созданный массив
y[1] = x[1]
for i = 2 to n
    y[i] = y[i - 1] ⊗ x[i]
return y
```

К сожалению, многопоточная версия SCAN далеко не очевидна. Например, замена цикла **for** циклом **parallel for** создаст гонки, поскольку каждая итерация тела цикла зависит от предыдущей. Приведенная далее процедура P-SCAN-1 выполняет параллельное вычисление $\otimes$-префикса, хотя и очень неэффективно.

```
P-SCAN-1(x)
n = x.length
y[1..n] — вновь созданный массив
P-SCAN-1-AUX(x, y, 1, n)
return y
```

```
P-SCAN-1-AUX(x, y, i, j)
parallel for l = i to j
    y[l] = P-REDUCE(x, 1, l)
```

б. Проанализируйте работу, интервал и уровень параллелизма процедуры P-SCAN-1.

Используя вложенный параллелизм, можно добиться более эффективного вычисления $\otimes$-префикса.

```
P-SCAN-2(x)
n = x.length
y[1..n] — вновь созданный массив
P-SCAN-2-AUX(x, y, 1, n)
return y
```

```
P-SCAN-2-AUX(x, y, i, j)
if i == j
    y[i] = x[i]
else k = ⌊(i + j)/2⌋
    spawn P-SCAN-2-AUX(x, y, i, k)
    P-SCAN-2-AUX(x, y, k + 1, j)
    sync
    parallel for l = k + 1 to j
        y[l] = y[k] ⊗ y[l]
```

в. Докажите, что процедура P-SCAN-2 корректна, и проанализируйте ее работу, интервал и уровень параллелизма.

Можно усовершенствовать как процедуру P-SCAN-1, так и процедуру P-SCAN-2, выполняя вычисление $\otimes$-префикса в два разных прохода по данным. При первом проходе мы собираем члены различных последовательных подмассивов x во временный массив t, а на втором проходе используем члены в t для вычисления окончательного результата y. Эта стратегия реализуется приведенным далее псевдокодом, в котором опущены некоторые выражения.

```
P-SCAN-3(x)
1  n = x.length
2  y[1..n] и t[1..n] — вновь созданные массивы
3  y[1] = x[1]
4  if n > 1
5      P-SCAN-UP(x, t, 2, n)
6      P-SCAN-DOWN(x[1], x, t, y, 2, n)
7  return y
```

```
P-SCAN-UP(x, t, i, j)
1  if i == j
2      return x[i]
3  else
4      k = ⌊(i + j)/2⌋
5      t[k] = spawn P-SCAN-UP(x, t, i, k)
6      right = P-SCAN-UP(x, t, k + 1, j)
7      sync
8      return ______              // Заполните пустое место
```

```
P-SCAN-DOWN(v, x, t, y, i, j)
1  if i == j
2      y[i] = v ⊗ x[i]
3  else
4      k = ⌊(i + j)/2⌋
       // В двух следующих строках заполните пустые места
5      spawn P-SCAN-DOWN(______, x, t, y, i, k)
6      P-SCAN-DOWN(______, x, t, y, k + 1, j)
7      sync
```

г. Заполните три пропущенные выражения в строке 8 процедуры P-SCAN-UP и в строках 5 и 6 процедуры P-SCAN-DOWN. Докажите корректность процедуры P-SCAN-3 со своими выражениями. (*Указание:* докажите, что значение v, переданное в процедуру P-SCAN-DOWN(v, x, t, y, i, j), удовлетворяет условию $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

д. Проанализируйте работу, интервал и уровень параллелизма процедуры P-SCAN-3.

27.5. Многопоточное простое шаблонное вычисление

Информатика переполнена алгоритмами, требующими заполнения элементов массива значениями, зависящими от уже вычисленных соседних значений, а также от другой информации, не изменяющейся в процессе вычислений. Структура соседних элементов не изменяется в процессе вычислений и называется ***шаблоном*** (stencil). Например, в разделе 15.4 представлен шаблонный алгоритм для

вычисления наидлиннейшей общей подпоследовательности, где значение элемента $c[i, j]$ зависит только от значений элементов $c[i-1, j]$, $c[i, j-1]$ и $c[i-1, j-1]$, а также от элементов x_i и y_j в двух входных последовательностях. Входные последовательности фиксированы, но алгоритм заполняет двумерный массив c таким образом, что элемент $c[i, j]$ вычисляется после вычисления трех элементов: $c[i-1, j]$, $c[i, j-1]$ и $c[i-1, j-1]$.

В данной задаче мы рассмотрим использование вложенного параллелизма для многопоточного шаблонного вычисления в массиве A размером $n \times n$, при котором значение, помещаемое в элемент $A[i, j]$, зависит только от значений $A[i', j']$, где $i' \leq i$ и $j' \leq j$ (и конечно, $i' \neq i$ или $j' \neq j$). Другими словами, значение элемента зависит только от значений элементов, находящихся выше или левее него, а также от статической информации, находящейся вне этого массива. Кроме того, в этой задаче мы полагаем, что если заполнены все элементы, от которых зависит элемент $A[i, j]$, то сам элемент $A[i, j]$ можно заполнить за время $\Theta(1)$ (как в процедуре LCS-LENGTH из раздела 15.4).

Можно разбить массив A размером $n \times n$ на четыре подмассива размером $n/2 \times n/2$ следующим образом:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} . \tag{27.11}$$

Теперь заметим, что подмассив A_{11} можно заполнить рекурсивно, поскольку он не зависит от элементов остальных трех подмассивов. После заполнения подмассива A_{11} можно продолжить рекурсивное заполнение подмассивов A_{12} и A_{21}, поскольку хотя они оба и зависят от A_{11}, но не зависят один от другого. В конце мы можем рекурсивно заполнить подмассив A_{22}.

а. Запишите многопоточный псевдокод SIMPLE-STENCIL, выполняющий это простое шаблонное вычисление с использованием алгоритма “разделяй и властвуй”, основанного на разбиении (27.11) и рассмотренного выше. (Не беспокойтесь о деталях базового случая, которые зависят от конкретного шаблона.) Запишите и решите рекуррентное соотношение для работы и интервала этого алгоритма в терминах n. Каков уровень его параллелизма?

б. Модифицируйте свое решение п. (а), разделив массив размером $n \times n$ на девять подмассивов размером $n/3 \times n/3$ и выполнив рекурсивные вычисления с максимально возможным параллелизмом. Проанализируйте свой алгоритм. Насколько увеличился или уменьшился уровень параллелизма по сравнению с алгоритмом из части (а)?

в. Обобщите свое решение пп. (а) и (б) следующим образом. Выберите целочисленное значение $b \geq 2$. Разделите массив размером $n \times n$ на b^2 подмассивов, каждый размером $n/b \times n/b$, с максимально возможной параллельностью рекурсивных вычислений. Чему равны работа, интервал и уровень параллелизма вашего алгоритма, выраженные через n и b? Докажите, что при использовании этого подхода уровень параллелизма должен составлять $o(n)$ при любом

выборе $b \geq 2$. (*Указание:* для последнего доказательства покажите, что степень n в выражении для уровня параллелизма строго меньше 1 при любом выборе $b \geq 2$.)

г. Запишите псевдокод многопоточного алгоритма для данного простого шаблонного вычисления, который достигает уровня параллелизма, равного $\Theta(n/\lg n)$. Докажите с помощью понятий работы и интервала, что данная задача фактически имеет присущий ей уровень параллелизма $\Theta(n)$. Как оказывается, достичь этого максимального уровня параллелизма нашему многопоточному псевдокоду не позволяет его природа "разделяй и властвуй".

27.6. Рандомизированные многопоточные алгоритмы

Как и в случае обычных последовательных алгоритмов, иногда требуется реализовать рандомизированные многопоточные алгоритмы. В данной задаче рассматривается, как адаптировать различные меры производительности для работы с ожидаемым поведением таких алгоритмов. В ней также требуется разработать и проанализировать многопоточный алгоритм рандомизированной быстрой сортировки.

а. Поясните, как изменить закон работы (27.2), закон интервала (27.3) и границы жадного планировщика (27.4) для работы с математическими ожиданиями случайных величин T_P, T_1 и T_∞.

б. Рассмотрим рандомизированный многопоточный алгоритм, в котором 1% времени мы имеем $T_1 = 10^4$ и $T_{10\,000} = 1$, но в течение 99% времени $T_1 = T_{10\,000} = 10^9$. Докажите, что ***ускорение*** рандомизированного многопоточного алгоритма следует определять как $\mathrm{E}[T_1]/\mathrm{E}[T_P]$, а не как $\mathrm{E}[T_1/T_P]$.

в. Докажите, что ***параллелизм*** рандомизированного многопоточного алгоритма следует определять как отношение $\mathrm{E}[T_1]/\mathrm{E}[T_\infty]$.

г. Превратите алгоритм RANDOMIZED-QUICKSORT, приведенный на с. 208, в многопоточный с применением вложенного параллелизма. (Не распараллеливайте RANDOMIZED-PARTITION.) Запишите псевдокод своего алгоритма P-RANDOMIZED-QUICKSORT.

д. Проанализируйте свой многопоточный алгоритм рандомизированной быстрой сортировки. (*Указание:* обратитесь к анализу алгоритма RANDOMIZED-SELECT на с. 246.)

Заключительные замечания

Параллельные компьютеры, их модели и алгоритмические модели параллельного программирования в процессе развития принимали различные формы. Предыдущие издания этой книги включали материал о сортирующих сетях и мо-

дели PRAM (Parallel Random-Access Machine, — параллельная машина с произвольным доступом). Модель параллельных данных [47, 167] является еще одной популярной алгоритмической моделью программирования, использующей в качестве примитивов векторы и матрицы.

Грэхэм (Graham) [148] и Брент (Brent) [54] показали, что существуют планировщики, достигающие границы из теоремы 27.1. Игер (Eager), Захорян (Zahorjan) и Лазовска (Lazowska) [97] показали, что любой жадный планировщик достигает этой границы, и предложили методологию использования работы и интервала (хотя и с другими терминами) для анализа параллельных алгоритмов. Блеллох (Blelloch) [46] разработал алгоритмическую модель программирования, основанную на работе и интервале (который он называл "глубиной" вычисления) для программирования с параллельными данными. Блюмоф (Blumofe) и Лейзерсон (Leiserson) [51] разработали распределенный алгоритм планирования для динамической многопоточности, основанный на рандомизированном "хищении работы" (work-stealing), и показал, что он достигает границы $\mathrm{E}[T_P] \le T_1/P + O(T_\infty)$. Арора (Arora), Блюмоф (Blumofe) и Плакстон (Plaxton) [19], а также Блеллох (Blelloch), Гиббонс (Gibbons) и Матиас (Matias) [48] также нашли доказуемо хорошие алгоритмы для планирования вычислений с динамической многопоточностью.

На многопоточный псевдокод и модель программирования большое влияние оказали проект Cilk [50,117] из MIT и расширения Cilk++ [70] для языка программирования С++ от Cilk Arts, Inc. Многие многопоточные алгоритмы из этой главы взяты из неопубликованных лекций Ч.Э. Лейзерсона (C.E. Leiserson) и Г. Прокопа (H. Prokop) и были реализованы на Cilk или Cilk++. Многопоточный алгоритм сортировки слиянием разработан под влиянием алгоритма Акла (Akl) [12].

Понятие последовательной согласованности предложено Лампортом (Lamport) [222].

Глава 28. Работа с матрицами

Поскольку операции над матрицами являются сердцем научных расчетов, эффективные алгоритмы для работы с матрицами представляют значительный практический интерес. В этой главе делается особый упор на задачу умножения матриц и решения систем линейных уравнений. Основы теории матриц излагаются в приложении Г.

В разделе 28.1 речь идет о решении систем линейных уравнений с использованием LUP-разложения. Затем в разделе 28.2 изучается тесная взаимосвязь между умножением и обращением матриц. Наконец в разделе 28.3 рассматриваются важный класс симметричных положительно определенных матриц и их использование для решения переопределенных систем линейных уравнений методом наименьших квадратов.

Одним из важнейших вопросов, возникающих на практике, является ***численная устойчивость*** (numerical stability). Из-за ограниченной точности представления действительных чисел в реальном компьютере в процессе вычислений могут резко нарастать ошибки округления, что приводит к неверным результатам. Такие вычисления называются ***численно неустойчивыми*** (numerically unstable). Несмотря на важность данного вопроса, мы лишь поверхностно коснемся его в данной главе, так что мы рекомендуем читателям обратиться к отличной книге Голуба (Golub) и ван Лоана (van Loan) [143], в которой детально рассматриваются вопросы численной устойчивости.

28.1. Решение систем линейных уравнений

Во многих приложениях требуется решать системы линейных уравнений. Систему линейных уравнений можно сформулировать в виде матричного уравнения, в котором каждая матрица или вектор принадлежат полю, обычно полю действительных чисел $\mathbb{R}$. В этом разделе рассматривается решение систем линейных уравнений с помощью метода, называемого LUP-разложением.

Начнем с системы линейных уравнений с n неизвестными $x_1, x_2, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \tag{28.1}$$

Решением уравнений (28.1) является множество значений $x_1, x_2, \ldots, x_n$, которое удовлетворяет всем уравнениям одновременно. В этом разделе мы рассматриваем только случай, когда имеется ровно n уравнений с n неизвестными.

Можно удобно переписать уравнения (28.1) в виде матрично-векторного уравнения

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

или, полагая $A = (a_{ij})$, $x = (x_i)$ и $b = (b_i)$, в эквивалентной форме как

$$Ax = b . \tag{28.2}$$

Если матрица A невырождена, имеется обратная к ней матрица A^{-1}, и

$$x = A^{-1}b \tag{28.3}$$

является вектором решения. Можно доказать, что x является единственным решением уравнения (28.2), следующим образом. Если имеется два решения, x и x', то $Ax = Ax' = b$ и, обозначая тождественную матрицу как I,

$$\begin{aligned} x &= Ix \\ &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' . \end{aligned}$$

В этом разделе мы в основном будем иметь дело с невырожденной матрицей A, или, что эквивалентно (согласно теореме Г.1), с матрицей A, ранг которой равен количеству n неизвестных. Существуют, однако, и другие варианты, которые заслуживают краткого рассмотрения. Если количество уравнений меньше количества n неизвестных (или, говоря более обобщенно, если ранг матрицы A меньше n), то система линейных уравнений является ***недоопределенной*** (underdetermined). Обычно недоопределенная система линейных уравнений имеет

бесконечно много решений, хотя может не иметь решений вовсе, если уравнения системы оказываются несовместимыми. Если количество уравнений превышает количество n неизвестных, система линейных уравнений является ***переопределенной*** (overdetermined), и у нее может не существовать решений. В разделе 28.3 рассматривается важная задача поиска хороших приближенных решений переопределенных систем линейных уравнений.

Вернемся к нашей задаче решения системы линейных уравнений $Ax = b$ из n уравнений от n неизвестных. Мы можем вычислить A^{-1}, а затем, воспользовавшись уравнением (28.3), умножить b на A^{-1} и получить $x = A^{-1}b$. На практике этого подхода избегают из-за его численной неустойчивости. К счастью, другой подход — LUP-разложение — численно устойчив и обладает тем преимуществом, что на практике оказывается более быстрым.

Обзор LUP-разложения

Идея, лежащая в основе LUP-разложения, состоит в поиске трех матриц, L, U и P, размером $n \times n$, таких, что

$$PA = LU , \tag{28.4}$$

где

- L — единичная нижнетреугольная матрица;
- U — верхнетреугольная матрица;
- P — матрица перестановки.

Матрицы L, U и P, удовлетворяющие уравнению (28.4), называются ***LUP-разложением*** (LUP decomposition) матрицы A. Мы покажем, что всякая невырожденная матрица A допускает такое разложение.

Преимущество вычисления LUP-разложения матрицы A основано на том, что система линейных уравнений решается гораздо легче, если ее матрица треугольна, что и выполняется в случае матриц L и U. Найдя LUP-разложение матрицы A, мы можем решить уравнение (28.2), $Ax = b$, путем решения треугольной системы линейных уравнений, как показано далее. Умножая обе части уравнения $Ax = b$ на P, мы получим эквивалентное уравнение $PAx = Pb$, которое в соответствии с упражнением Г.1.4 эквивалентно перестановке уравнений из (28.1). Используя разложение (28.4), получаем

$$LUx = Pb .$$

Теперь можно решить полученное уравнение, решив две треугольные системы линейных уравнений. Обозначим $y = Ux$, где x — решение исходной системы линейных уравнений. Сначала решим нижнетреугольную систему линейных уравнений

$$Ly = Pb , \tag{28.5}$$

найдя неизвестный вектор y с помощью метода, который называется прямой подстановкой. После этого, имея вектор y, решим верхнетреугольную систему ли-

нейных уравнений

$$Ux = y \ , \tag{28.6}$$

найдя x с помощью обратной подстановки. Поскольку матрица перестановки P обращаема (см. упр. Г.2.3), умножение обеих частей уравнения (28.4) на P^{-1} дает $P^{-1}PA = P^{-1}LU$, так что

$$A = P^{-1}LU \ . \tag{28.7}$$

Следовательно, вектор x является искомым решением системы линейных уравнений $Ax = b$:

$$\begin{aligned}
Ax &= P^{-1}LUx && \text{(согласно (28.7))} \\
&= P^{-1}Ly && \text{(согласно (28.6))} \\
&= P^{-1}Pb && \text{(согласно (28.5))} \\
&= b \ .
\end{aligned}$$

Теперь рассмотрим, как работают прямая и обратная подстановки, а затем приступим к самой задаче вычисления LUP-разложения.

Прямая и обратная подстановки

Прямая подстановка (forward substitution) позволяет решить нижнетреугольную систему линейных уравнений (28.5) для данных L, P и b за время $\Theta(n^2)$. Для удобства мы представим перестановку P в компактной форме с помощью массива $\pi[1..n]$. Элемент $\pi[i]$ при $i = 1, 2, \ldots, n$ указывает, что $P_{i,\pi[i]} = 1$ и $P_{ij} = 0$ для $j \neq \pi[i]$. Таким образом, в матрице PA на пересечении i-й строки и j-го столбца находится элемент $a_{\pi[i],j}$, а i-м элементом Pb является $b_{\pi[i]}$. Поскольку L является единичной нижнетреугольной матрицей, уравнение (28.5) можно переписать следующим образом:

$$\begin{aligned}
y_1 \qquad\qquad\qquad\qquad &= b_{\pi[1]} \ , \\
l_{21}y_1 + \quad y_2 \qquad\qquad\quad &= b_{\pi[2]} \ , \\
l_{31}y_1 + l_{32}y_2 + \quad y_3 \qquad &= b_{\pi[3]} \ , \\
\\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]} \ .
\end{aligned}$$

Из первого уравнения получаем, что $y_1 = b_{\pi[1]}$. Зная значение y_1, его можно подставить во второе уравнение и найти

$$y_2 = b_{\pi[2]} - l_{21}y_1 \ .$$

Теперь можно подставить в третье уравнение два найденных значения, y_1 и y_2, и получить

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) \ .$$

В общем случае для поиска y_i мы подставляем найденные значения $y_1, y_2, \ldots, y_{i-1}$ в i-е уравнение и находим

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j \; .$$

После того как мы нашли y, найти x из уравнения (28.6) можно, воспользовавшись ***обратной подстановкой*** (back substitution), которая аналогична прямой. В этом случае мы решаем первым n-е уравнение и работаем в обратном направлении, к первому уравнению. Как и прямая подстановка, этот процесс требует времени $\Theta(n^2)$. Поскольку U является верхнетреугольной матрицей, систему линейных уравнений (28.6) можно переписать как

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 \; , \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 \; , \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} \; , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \; , \\
u_{n,n}x_n &= y_n \; .
\end{aligned}$$

Таким образом, можно последовательно найти $x_n, x_{n-1}, \ldots, x_1$ следующим образом:

$$\begin{aligned}
x_n &= y_n / u_{n,n} \; , \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \; , \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \; , \\
&\vdots
\end{aligned}$$

или, в общем случае,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} \; .$$

Процедура LUP-SOLVE для заданных P, L, U и b находит x путем комбинирования прямой и обратной подстановок. В псевдокоде предполагается, что размерность n хранится в атрибуте $L.rows$ и что матрица перестановки P представлена массивом π.

```
LUP-SOLVE(L, U, π, b)
1  n = L.rows
2  Пусть x и y — вновь созданные векторы длиной n
3  for i = 1 to n
4       y_i = b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
5  for i = n downto 1
6       x_i = (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
7  return x
```

Процедура LUP-SOLVE находит y в строках 3 и 4 с помощью прямой подстановки, а затем вычисляет x с помощью обратной подстановки в строках 5 и 6. Наличие внутри каждого цикла **for** неявного цикла суммирования приводит ко времени работы данной процедуры, равному $\Theta(n^2)$.

В качестве примера применения данных методов рассмотрим систему линейных уравнений

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

где

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} ,$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

и которую необходимо решить, найдя неизвестный вектор x. LUP-разложение матрицы A имеет вид

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} ,$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} ,$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} .$$

(Вы можете убедиться в корректности разложения, проверив, что $PA = LU$.) Используя прямую подстановку, находим y из $Ly = Pb$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

получив

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

путем вычисления сначала y_1, затем — y_2 и наконец — y_3. Затем с помощью обратной подстановки находим x из $Ux = y$:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

тем самым получив искомое решение исходной системы линейных уравнений

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

путем вычисления сначала x_3, затем — x_2 и наконец — x_1.

Вычисление LU-разложения

Мы показали, что если можно вычислить LUP-разложение невырожденной матрицы A, то для решения системы линейных уравнений $Ax = b$ можно воспользоваться простыми прямой и обратной подстановками. Осталось показать, как эффективно найти LUP-разложение матрицы A. Начнем со случая невырожденной матрицы A размером $n \times n$ и отсутствия матрицы P (или, что эквивалентно, $P = I_n$). В этом случае необходимо найти разложение $A = LU$. Матрицы L и U называются ***LU-разложением*** (LU decomposition) матрицы A.

Мы применим для построения LU-разложения процесс, известный как ***метод исключения Гаусса*** (Gauss elimination). Начнем с удаления первой переменной из всех уравнений, кроме первого, путем вычитания из этих уравнений первого, умноженного на соответствующий коэффициент. Затем та же операция будет проделана со вторым уравнением, чтобы в третьем и последующих уравнениях отсутствовали первая и вторая переменные. Процесс будет продолжаться до тех пор, пока мы не получим верхнетреугольную матрицу — это и будет искомая матрица U. Матрица L составляется из коэффициентов, участвовавших в процессе исключения переменных.

Мы реализуем описанную стратегию с помощью рекурсивного алгоритма. Итак, мы хотим построить LU-разложение невырожденной матрицы A разме-

ром $n \times n$. Если $n = 1$, задача решена, поскольку мы можем выбрать $L = I_1$ и $U = A$. При $n > 1$ разобьем A на четыре части:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

где $v = (v_2, v_3, \ldots, v_n) = (a_{21}, a_{22}, \ldots, a_{n1})$ представляет собой вектор-столбец длиной $(n-1)$, $w^{\mathrm{T}} = (w_2, w_3, \ldots, w_n)^{\mathrm{T}} = (a_{12}, a_{13}, \ldots, a_{1n})^{\mathrm{T}}$ является вектором-строкой длиной $(n-1)$, а A' — матрицей размером $(n-1) \times (n-1)$. Используя матричную алгебру (проверить полученный результат можно с помощью умножения), разложим матрицу A следующим образом:

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}. \quad (28.8)$$

Нули в первой и второй матрицах уравнения (28.8) представляют собой вектор-строку и вектор-столбец длиной $n-1$ с нулевыми элементами. Матрица vw^{T}/a_{11}, полученная тензорным произведением векторов v и w и делением каждого элемента полученной в результате умножения матрицы на a_{11}, представляет собой матрицу размером $(n-1) \times (n-1)$ (тот же размер имеет и матрица A', из которой она вычитается). Полученная в результате матрица размером $(n-1) \times (n-1)$

$$A' - vw^{\mathrm{T}}/a_{11} \quad (28.9)$$

называется ***дополнением Шура*** (Schur complement) матрицы A по отношению к элементу a_{11}.

Мы утверждаем, что если матрица A невырождена, то невырождено и дополнение Шура. Почему? Предположим, что дополнение Шура, представляющее собой матрицу размером $(n-1) \times (n-1)$, вырождено. Тогда по теореме Г.1 она имеет ранг, строго меньший $n-1$. Поскольку нижние $n-1$ элементов в первом столбце матрицы

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

равны 0, нижние $n-1$ строк должны иметь ранг, строго меньший $n-1$. Таким образом, ранг всей матрицы строго меньше n. Применив результат упр. Г.2.8 к уравнению (28.8), находим, что ранг матрицы A строго меньше n, так что согласно теореме Г.1 получаем противоречие с начальным условием невырожденности A.

Поскольку дополнение Шура невырождено, можно рекурсивно найти его LU-разложение. Запишем

$$A' - vw^{\mathrm{T}}/a_{11} = L'U' ,$$

где L' — единичная нижнетреугольная матрица, а U' — верхнетреугольная матрица. Тогда, прибегнув к матричной алгебре, получим

$$\begin{aligned}
A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU ,
\end{aligned}$$

что дает искомое LU-разложение. (Заметим, что поскольку матрица L' — единичная нижнетреугольная, таковой же является и матрица L; а поскольку матрица U' — верхнетреугольная, таковой же является и матрица U.)

Конечно, если $a_{11} = 0$, этот метод не работает, поскольку при этом должно быть выполнено деление на 0. Он также не работает, если верхний левый элемент дополнения Шура $A' - vw^{\mathrm{T}}/a_{11}$ равен нулю, поскольку в этом случае деление на нуль возникнет на следующем шаге рекурсии. Элементы, на которые в процессе LU-разложения выполняется деление, называются ***ведущими*** (pivots) и располагаются на диагонали матрицы U. Причина, по которой в LUP-разложение включается матрица перестановок P, состоит в том, чтобы избежать деления на нулевые элементы. Использование матрицы перестановки для того, чтобы избежать деления на нуль (или на малые величины, что вносит вклад в численную неустойчивость), называется ***выбором ведущего элемента*** (pivoting).

Важным классом матриц, для которых LU-разложение всегда работает корректно, является класс симметричных положительно определенных матриц. Такие матрицы не требуют выбора ведущего элемента, так что описанная стратегия может быть применена без опасения столкнуться с делением на нуль. Мы докажем это (а также некоторые другие утверждения) в разделе 28.3.

Наш код для LU-разложения матрицы A следует рекурсивной стратегии, хотя рекурсия в нем и заменена итеративным циклом (такое преобразование является стандартной оптимизацией процедуры с оконечной рекурсией, когда последней операцией процедуры является рекурсивный вызов ее самой; см. задачу 7.4). В коде предполагается, что размерность матрицы A хранится в атрибуте $A.rows$. Мы инициализируем матрицу U нулями ниже диагонали, а матрицу L единицами на диагонали и нулями — выше нее. Каждая итерация работает с квадратной подматрицей, используя ее верхний левый элемент в качестве ведущего для вычисления векторов v и w и дополнения Шура, которое становится квадратной подматрицей, обрабатываемой на следующей итерации.

```
LU-DECOMPOSITION(A)
1  n = A.rows
2  L и U являются новыми матрицами размером n × n
3  Инициализируем U нулями ниже диагонали
4  Инициализируем L единицами на диагонали и нулями выше нее
5  for k = 1 to n
6      u_kk = a_kk
7      for i = k + 1 to n
8          l_ik = a_ik / a_kk        // a_ik хранит v_i
9          u_ki = a_ki               // a_ki хранит w_i
10     for i = k + 1 to n
11         for j = k + 1 to n
12             a_ij = a_ij − l_ik u_kj
13 return L и U
```

Внешний цикл **for**, начинающийся в строке 5, выполняет каждый шаг рекурсии по одному разу. В теле этого цикла в строке 6 ведущим элементом полагается элемент $u_{kk} = a_{kk}$. В цикле **for** в строках 7–9 (который не выполняется, когда $k = n$) для обновления матриц U и L используются векторы v и w. В строке 8 определяются элементы L, находящиеся ниже диагонали, и выполняется сохранение v_i/a_{kk} в элементе l_{ik}; а в строке 9 вычисляются элементы U, расположенные над диагональю, и выполняется сохранение w_i в u_{ki}. И наконец в строках 10–12 вычисляются элементы дополнения Шура, которые затем сохраняются в матрице A (в строке 12 не нужно выполнять деление на a_{kk}, так как оно уже выполнено в строке 8 при вычислении l_{ik}). В связи с тройной вложенностью в циклы строки 12 время работы процедуры LU-DECOMPOSITION равно $\Theta(n^3)$.

На рис. 28.1 показана работа процедуры LU-DECOMPOSITION. Здесь проиллюстрирована стандартная оптимизация процедуры, при которой мы храним значащие элементы L и U без привлечения дополнительной памяти, непосредственно в матрице A. Иначе говоря, мы можем установить соответствие между каждым элементом a_{ij} и либо l_{ij} (если $i > j$), либо u_{ij} (если $i \leq j$) и обновлять матрицу A так, чтобы по окончании работы процедуры в ней хранились матрицы L и U. Чтобы получить псевдокод этой оптимизации из приведенного выше, необходимо просто заменить каждое обращение к l и u на обращение к a; можно легко убедиться в том, что такое преобразование сохраняет корректность алгоритма.

Вычисление LUP-разложения

В общем случае при решении системы линейных уравнений $Ax = b$ может оказаться, что необходимо выбирать ведущие элементы среди недиагональных элементов матрицы A для того, чтобы избежать деления на 0. Причем нежелательно не только деление на 0, но и просто на малое число, даже если матрица A невырождена, поскольку в результате можно получить численную неустойчивость. В связи с этим в качестве ведущего элемента следует выбирать наибольший возможный элемент.

2	3	1	5
6	13	5	19
2	19	10	23
4	10	11	31

(а)

2	3	1	5
3	4	2	4
1	16	9	18
2	4	9	21

(б)

2	3	1	5
3	4	2	4
1	4	1	2
2	1	7	17

(в)

2	3	1	5
3	4	2	4
1	4	1	2
2	1	7	3

(г)

$$\underbrace{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}_{U}$$

(д)

Рис. 28.1. Работа процедуры LU-DECOMPOSITION. **(а)** Матрица A. **(б)** Элемент $a_{11} = 2$ (в черном круге) является ведущим, заштрихованный столбец представляет собой v/a_{11}, а заштрихованная строка — w^{T}. Вычисленные к этому моменту элементы U находятся над горизонтальной линией, а элементы L — слева от вертикальной линии. Матрица дополнения Шура $A' - vw^{\mathrm{T}}/a_{11}$ занимает нижнюю правую часть. **(в)** Теперь работа продолжается с матрицей дополнения Шура из части (б). Элемент $a_{22} = 4$ в черном круге является ведущим, а заштрихованные столбец и строка представляют собой v/a_{22} и w^{T} (в разбиении дополнения Шура) соответственно. Линии делят матрицу на вычисленные к этому моменту элементы U (вверху), элементы L (слева) и новое дополнение Шура (внизу справа). **(г)** После следующего шага матрица A разложена. (Элемент 3 в новом дополнении Шура становится частью U при завершении рекурсии.) **(д)** Разложение $A = LU$.

Математические основы LUP-разложения аналогичны LU-разложению. Вспомним, что имеется невырожденная матрица A размером $n \times n$ и требуется найти матрицу перестановки P, единичную нижнетреугольную матрицу L и верхнетреугольную матрицу U, такие, что $PA = LU$. Перед разделением матрицы A на части, как при вычислении LU-разложения, мы перемещаем ненулевой элемент, скажем, a_{k1}, откуда-то из первого столбца матрицы в позицию $(1,1)$ (если первый столбец содержит только нулевые значения, то матрица вырождена, поскольку ее определитель равен 0 (см. теоремы Г.4 и Г.5)). Для сохранения множества исходных линейных уравнений мы меняем местами строки 1 и k, что эквивалентно умножению матрицы A на матрицу перестановки Q слева (см. упр. Г.1.4). Тогда мы можем записать QA как

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} ,$$

где $v = (a_{21}, a_{31}, \ldots, a_{n1})$, за исключением того, что a_{11} заменяет a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})^{\mathrm{T}}$; а A' является матрицей размером $(n-1) \times (n-1)$. Поскольку $a_{k1} \neq 0$, можно выполнить почти те же преобразования, что и при LU-разложе-

нии, но теперь с гарантией, что деления на нуль не будет:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} . \end{aligned}$$

Как мы видели в LU-разложении, если матрица A невырождена, то невырождено и дополнение Шура $A' - vw^{\mathrm{T}}/a_{k1}$. Таким образом, мы можем индуктивно найти его LUP-разложение, дающее единичную нижнетреугольную матрицу L', верхнетреугольную матрицу U' и матрицу перестановки P', такие, что

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U' .$$

Определим матрицу

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q ,$$

которая является матрицей перестановки в силу того, что она представляет собой произведение двух матриц перестановки (см. упр. Г.1.4). Мы имеем

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU , \end{aligned}$$

что и дает искомое LUP-разложение. Поскольку L' — единичная нижнетреугольная матрица, такой же будет и L, и поскольку U' — верхнетреугольная матрица, такой же будет и U. Заметим, что в отличие от LU-разложения, на матрицу перестановки P' должны умножаться и вектор-столбец v/a_{k1}, и дополнение Шура $A' - vw^{\mathrm{T}}/a_{k1}$. Вот как выглядит псевдокод LUP-разложения.

LUP-DECOMPOSITION(A)

```
1  n = A.rows
2  π[1..n] — вновь созданный массив
3  for i = 1 to n
4      π[i] = i
5  for k = 1 to n
6      p = 0
7      for i = k to n
8          if |a_ik| > p
9              p = |a_ik|
10             k' = i
11     if p == 0
12         error "вырожденная матрица"
13     Обменять π[k] с π[k']
14     for i = 1 to n
15         Обменять a_ki с a_k'i
16     for i = k + 1 to n
17         a_ik = a_ik / a_kk
18         for j = k + 1 to n
19             a_ij = a_ij − a_ik a_kj
```

Как и в процедуре LU-DECOMPOSITION, в псевдокоде LUP-разложения LUP-DECOMPOSITION рекурсия заменяется итерацией. В качестве усовершенствования непосредственной реализации рассмотренной рекурсии мы динамически поддерживаем матрицу перестановки P в виде массива π, где $\pi[i] = j$ означает, что i-я строка массива P содержит 1 в столбце j. Мы также реализуем код, который вычисляет L и U "на месте", в матрице A, т.е. по окончании работы процедуры

$$a_{ij} = \begin{cases} l_{ij} & \text{если } i > j\ , \\ u_{ij} & \text{если } i \leq j\ . \end{cases}$$

На рис. 28.2 показано, как процедура LUP-DECOMPOSITION выполняет разложение матрицы. В строках 3 и 4 выполняется инициализация массива π, который после этого представляет тождественную перестановку. Внешний цикл **for**, начинающийся в строке 5, реализует рекурсию. При каждой итерации внешнего цикла строки 6–10 определяют элемент $a_{k'k}$ с наибольшим абсолютным значением в текущем первом столбце (столбец k) матрицы размером $(n-k+1) \times (n-k+1)$, LUP-разложение которой мы ищем. Если все элементы в текущем первом столбце нулевые, строки 11 и 12 сообщают о вырожденности матрицы. Чтобы получить ведущий элемент, мы обмениваем $\pi[k']$ с $\pi[k]$ в строке 13, а также обмениваем строки k и k' матрицы A в строках 14 и 15 процедуры, тем самым делая ведущим элемент a_{kk}. (Обмен строк выполняется полностью, поскольку в выводе данного метода выше на P' умножается не только $A' - vw^{\mathrm{T}}/a_{k1}$, но и v/a_{k1}.) Наконец в строках 16–19 вычисляется дополнение Шура, практически так же, как и в стро-

1	2	0	2	0.6
2	3	3	4	−2
3	5	5	4	2
4	−1	−2	3.4	−1

(а)

3	5	5	4	2
2	3	3	4	−2
1	2	0	2	0.6
4	−1	−2	3.4	−1

(б)

3	5	5	4	2
2	0.6	0	1.6	−3.2
1	0.4	−2	0.4	−.2
4	−0.2	−1	4.2	−0.6

(в)

3	5	5	4	2
2	0.6	0	1.6	−3.2
1	0.4	−2	0.4	−0.2
4	−0.2	−1	4.2	−0.6

(г)

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	−1	4.2	−0.6

(д)

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	0.5	4	−0.5

(е)

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	0.5	4	−0.5

(ж)

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	0.6	0	1.6	−3.2

(з)

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	0.6	0	0.4	−3

(и)

$$\underbrace{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}}_{U}$$

(к)

Рис. 28.2. Работа процедуры LUP-DECOMPOSITION. **(а)** Входная матрица A с тождественной перестановкой строк слева. Шаг 1 алгоритма находит, что элемент 5 (в черном круге в третьей строке) является ведущим для первого столбца. **(б)** Выполняются обмен строк 1 и 3 и обновление матрицы перестановки. Заштрихованные столбец и строка представляют v и w^{T}. **(в)** Вектор v заменяется на $v/5$, и нижняя правая часть матрицы заменяется дополнением Шура. Линии делят матрицу на три области: элементы U (вверху), элементы L (слева) и элементы дополнения Шура (внизу справа). **(г)–(е)** Шаг 2. **(ж)–(и)** Шаг 3. На шаге 4 (последнем) не происходит никаких изменений. **(к)** LUP-разложение $PA = LU$.

ках 7–12 процедуры LU-DECOMPOSITION, с тем отличием, что здесь результаты работы записываются в матрицу A, без привлечения дополнительной памяти.

В силу тройной вложенности циклов время работы процедуры LUP-DECOMPOSITION равно $\Theta(n^3)$, т.е. оно точно такое же, как и в случае процедуры LU-DECOMPOSITION. Следовательно, выбор ведущего элемента приводит к увеличению времени работы не более чем на постоянный множитель.

Упражнения

28.1.1

Решите систему линейных уравнений

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

с помощью прямой подстановки.

28.1.2

Найдите LU-разложение матрицы

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} .$$

28.1.3

Решите систему линейных уравнений

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

с помощью LUP-разложения.

28.1.4

Как выглядит LUP-разложение диагональной матрицы?

28.1.5

Как найти LUP-разложение матрицы перестановки? Докажите его единственность.

28.1.6

Покажите, что для всех $n \geq 1$ существует вырожденная матрица размером $n \times n$, имеющая LU-разложение.

28.1.7

Необходимо ли выполнение тела внешнего цикла **for** в процедуре LU-DECOMPOSITION при $k = n$? А в процедуре LUP-DECOMPOSITION?

28.2. Обращение матриц

Хотя на практике для решения систем линейных уравнений обращение матриц обычно не используется, а вместо этого применяются другие, численно более

устойчивые, методы, например LUP-разложение, иногда все же требуется вычислить обратную матрицу. В этом разделе мы покажем, что для обращения матриц можно использовать уже рассмотренный нами метод LUP-разложения. Мы также докажем, что умножение матриц и обращение матрицы — задачи одинаковой сложности, так что для решения одной задачи мы можем использовать алгоритм для решения другой, получив при этом одинаковое асимптотическое время работы. В частности, для обращения матриц мы можем применить алгоритм Штрассена из раздела 4.2 (к слову, в своей работе Штрассен преследовал цель ускорить решение систем линейных уравнений).

Вычисление обратной матрицы из LUP-разложения

Предположим, что имеется LUP-разложение матрицы A на три матрицы, L, U и P, такие, что $PA = LU$. Используя процедуру LUP-SOLVE, мы можем решить уравнение вида $Ax = b$ за время $\Theta(n^2)$. Поскольку LUP-разложение зависит только от A, но не от b, мы можем использовать ту же процедуру LUP-SOLVE для решения другой системы линейных уравнений вида $Ax = b'$ за дополнительное время $\Theta(n^2)$. Обобщая, имея LUP-разложение матрицы A, мы можем решить k систем линейных уравнений $Ax = b$, отличающихся только свободными членами b, за время $\Theta(kn^2)$.

Уравнение

$$AX = I_n \; , \tag{28.10}$$

определяющее матрицу X, обратную A, можно рассматривать как множество из n различных систем линейных уравнений вида $Ax = b$. Эти уравнения позволяют найти матрицу X, обратную матрице A. Более строго, обозначим i-й столбец X через X_i и вспомним, что i-м столбцом матрицы I_n является единичный вектор e_i. Мы можем найти X в уравнении (28.10), использовав LUP-разложение для решения набора уравнений

$$AX_i = e_i$$

для каждого X_i в отдельности. При наличии LUP-разложения поиск каждого столбца X_i требует времени $\Theta(n^2)$, так что полное время вычисления обратной матрицы X на основе LUP-разложения исходной матрицы A требует времени $\Theta(n^3)$. Поскольку LUP-разложение A также вычисляется за время $\Theta(n^3)$, задача вычисления обратной к A матрицы A^{-1} решается за время $\Theta(n^3)$.

Умножение матриц и обращение матрицы

Теперь покажем, каким образом можно использовать ускоренное умножение матриц для ускорения обращения матрицы (это ускорение имеет скорее теоретический интерес, чем практическое применение). В действительности мы докажем более строгое утверждение — умножение матриц эквивалентно обращению матрицы в следующем смысле. Если обозначить через $M(n)$ время, необходимое для умножения двух матриц размером $n \times n$, то можно обратить невырожденную матрицу размером $n \times n$ за время $O(M(n))$. Кроме того, если обозначить через $I(n)$ время, необходимое для обращения матрицы размером $n \times n$, то можно

перемножить две матрицы размером $n \times n$ за время $O(I(n))$. Мы докажем эти утверждения как две отдельные теоремы.

Теорема 28.1 (Умножение не сложнее обращения)

Если можно обратить матрицу размером $n \times n$ за время $I(n)$, где $I(n) = \Omega(n^2)$ и удовлетворяет условию регулярности $I(3n) = O(I(n))$, то две матрицы размером $n \times n$ можно перемножить за время $O(I(n))$.

Доказательство. Пусть A и B представляют собой матрицы размером $n \times n$, произведение которых C необходимо вычислить. Определим матрицу D размером $3n \times 3n$ как

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} .$$

Обратной к матрице D является

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix} ,$$

так что мы можем вычислить произведение AB, просто взяв верхнюю правую подматрицу размером $n \times n$ из матрицы D^{-1}.

Матрицу D можно построить за время $\Theta(n^2)$, которое представляет собой $O(I(n))$, поскольку мы считаем, что $I(n) = \Omega(n^2)$, и обратить ее за время $O(I(3n)) = O(I(n))$ согласно условию регулярности, накладываемому на $I(n)$. Таким образом, $M(n) = O(I(n))$. ∎

Заметим, что $I(n) = \Theta(n^c \lg^d n)$ удовлетворяет условию регулярности при любых константах $c > 0$ и $d \geq 0$.

Доказательство того, что обращение матрицы не сложнее умножения, опирается на некоторые свойства симметричных положительно определенных матриц, которые мы докажем в разделе 28.3.

Теорема 28.2 (Обращение не сложнее умножения)

Предположим, что мы можем умножить две действительные матрицы размером $n \times n$ за время $M(n)$, где $M(n) = \Omega(n^2)$ и, кроме того, удовлетворяет условиям регулярности $M(n+k) = O(M(n))$ для произвольного $0 \leq k \leq n$ и $M(n/2) \leq cM(n)$ для некоторой константы $c < 1/2$. В таком случае мы можем обратить любую действительную невырожденную матрицу размером $n \times n$ за время $O(M(n))$.

Доказательство. Здесь данная теорема доказывается для действительных чисел. В упр. 28.2.6 предлагается обобщить ее для комплексных матриц.

Можно считать, что n является точной степенью 2, поскольку мы имеем

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

для любого $k > 0$. Таким образом, выбрав k так, чтобы величина $n + k$ была степенью 2, мы увеличиваем исходную матрицу до размера, представляющего собой степень 2, а искомую обратную матрицу A^{-1} получаем как часть обращенной матрицы большего размера. Первое условие регулярности $M(n)$ гарантирует, что такое увеличение не вызовет увеличения времени работы более чем на постоянный множитель.

Предположим теперь, что матрица A имеет размер $n \times n$ и является симметричной и положительно определенной. Разобьем матрицу A и обратную к ней A^{-1} на четыре подматрицы размером $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \quad \text{и} \quad A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} . \tag{28.11}$$

Обозначив через

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.12}$$

дополнение Шура матрицы A по отношению к подматрице B (информацию об этом виде дополнений Шура можно найти в разделе 28.3), имеем

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \tag{28.13}$$

поскольку $AA^{-1} = I_n$, как вы можете убедиться, перемножив матрицы. Матрица A симметрична и положительно определена, поэтому из лемм 28.4 и 28.5 из раздела 28.3 вытекает, что и B, и S симметричны и положительно определены. Таким образом, согласно лемме 28.3 из раздела 28.3 существуют обратные матрицы B^{-1} и S^{-1}, а согласно упр. Г.2.6 B^{-1} и S^{-1} симметричны, так что $(B^{-1})^{\mathrm{T}} = B^{-1}$ и $(S^{-1})^{\mathrm{T}} = S^{-1}$. Итак, мы можем вычислить подматрицы R, T, U и V матрицы A^{-1} описанным далее способом (где все упоминаемые матрицы имеют размер $n/2 \times n/2$).

1. Образуем подматрицы B, C, C^{T} и D матрицы A.
2. Рекурсивно вычислим обратную матрице B матрицу B^{-1}.
3. Вычислим произведение матриц $W = CB^{-1}$, а затем транспонируем его W^{T}, получив матрицу, эквивалентную $B^{-1}C^{\mathrm{T}}$ (согласно упр. Г.1.2 и $(B^{-1})^{\mathrm{T}} = B^{-1}$).
4. Вычислим сначала произведение матриц $X = WC^{\mathrm{T}}$, эквивалентное $CB^{-1}C^{\mathrm{T}}$, а затем — матрицу $S = D - X = D - CB^{-1}C^{\mathrm{T}}$.
5. Рекурсивно вычислим обратную матрицу S^{-1} и присвоим ее матрице V.

6. Вычислим произведение матриц $Y = S^{-1}W$, равное $S^{-1}CB^{-1}$, а затем транспонируем его Y^{T}, что равно $B^{-1}C^{\mathrm{T}}S^{-1}$ (согласно упр. Г.1.2 $(B^{-1})^{\mathrm{T}} = B^{-1}$ и $(S^{-1})^{\mathrm{T}} = S^{-1}$). Присвоим T значение $-Y^{\mathrm{T}}$, а U — значение $-Y$.
7. Вычислим произведение матриц $Z = W^{\mathrm{T}}Y$, равное $B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}$, и присвоим R значение $B^{-1} + Z$.

Таким образом, можно инвертировать симметричную положительно определенную матрицу размером $n \times n$ путем обращения двух матриц размером $n/2 \times n/2$ в пп. 2 и 5, последующего выполнения четырех перемножений матриц размером $n/2 \times n/2$ в пп. 3, 4, 6 и 7, а также выполнения дополнительных действий по извлечению подматриц из A, вставке подматриц в A^{-1} ценой $O(n^2)$, и константного количества сложений, вычитаний и транспонирований матриц размером $n/2 \times n/2$. В результате мы получаем следующее рекуррентное соотношение:

$$\begin{aligned} I(n) &\leq 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) \ . \end{aligned}$$

Вторая строка выполняется, поскольку из второго условия регулярности в формулировке теоремы вытекает, что $4M(n/2) < 2M(n)$, и поскольку мы предполагаем, что $M(n) = \Omega(n^2)$. Третья строка следует из того, что второе условие регулярности позволяет применить случай 3 основной теоремы (теоремы 4.1).

Остается доказать, что асимптотическое время умножения матриц может быть получено для обращения невырожденной матрицы A, которая не является симметричной и положительно определенной. Основная идея заключается в том, что для любой невырожденной матрицы A матрица $A^{\mathrm{T}}A$ симметричная (согласно упр. Г.1.2) и положительно определенная (в соответствии с теоремой Г.6). Все, что остается, — это привести задачу обращения матрицы A к задаче обращения матрицы $A^{\mathrm{T}}A$.

Такое приведение основано на наблюдении, что если A является невырожденной матрицей размером $n \times n$, то

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}} \ ,$$

поскольку $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$, а обратная матрица единственна. Следовательно, можно вычислить A^{-1}, сначала умножив A^{T} на A для получения симметричной положительно определенной матрицы $A^{\mathrm{T}}A$, а затем обратив эту матрицу с помощью описанного выше рекурсивного алгоритма и умножив полученный результат на A^{T}. Каждый из перечисленных шагов требует времени $O(M(n))$, так что обращение любой невырожденной матрицы с действительными элементами может быть выполнено за время $O(M(n))$. ■

Доказательство теоремы 28.2 наводит на мысль о том, каким образом можно решать систему уравнений $Ax = b$ с невырожденной матрицей A с помощью LU-разложения, без выбора ведущего элемента. Для этого обе части уравнения нужно умножить на A^{T}, получив уравнение $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. Такое преобразова-

ние не влияет на x в силу обращаемости матрицы A^{T}, а тот факт, что матрица $A^{\mathrm{T}}A$ является симметричной положительно определенной матрицей, позволяет использовать для решения метод LU-разложения. После этого применение прямой и обратной подстановок с использованием правой части уравнения, равной $A^{\mathrm{T}}b$, дает решение x исходного уравнения. Хотя теоретически этот метод вполне корректен, на практике процедура LUP-DECOMPOSITION работает существенно лучше. Во-первых, она требует меньшего количества арифметических операций (отличающегося постоянным множителем от количества операций в описанном методе), а во-вторых, численная устойчивость LUP-DECOMPOSITION несколько выше.

Упражнения

28.2.1

Пусть $M(n)$ — время, необходимое для умножения двух матриц размером $n \times n$, а $S(n)$ — время, необходимое для возведения матрицы размером $n \times n$ в квадрат. Покажите, что умножение матриц и возведение матрицы в квадрат имеют одну и ту же сложность: из $M(n)$-алгоритма умножения матриц вытекает $O(M(n))$-алгоритм возведения матрицы в квадрат, а $S(n)$-алгоритм возведения матрицы в квадрат дает $O(S(n))$-алгоритм умножения матриц.

28.2.2

Пусть $M(n)$ представляет собой время, необходимое для умножения двух матриц размером $n \times n$. Покажите, что из существования алгоритма умножения матриц со временем работы $M(n)$ вытекает существование алгоритма LUP-разложения со временем работы $O(M(n))$.

28.2.3

Пусть $M(n)$ представляет собой время, необходимое для умножения двух матриц размером $n \times n$, а через $D(n)$ обозначим время, необходимое для поиска детерминанта матрицы размером $n \times n$. Покажите, что умножение матриц и вычисление детерминанта, по сути, имеют одинаковую сложность: из алгоритма умножения матриц со временем работы $M(n)$ вытекает алгоритм поиска детерминанта за время $O(M(n))$, а из алгоритма вычисления детерминанта за время $D(n)$ вытекает алгоритм умножения матриц за время $O(D(n))$.

28.2.4

Пусть $M(n)$ представляет собой время, необходимое для умножения двух булевых матриц размером $n \times n$, а $T(n)$ — время, необходимое для поиска транзитивного замыкания булевой матрицы размером $n \times n$. (См. раздел 25.2.) Покажите, что из алгоритма умножения булевых матриц со временем работы $M(n)$ вытекает алгоритм поиска транзитивного замыкания со временем работы $O(M(n) \lg n)$, а из алгоритма поиска транзитивного замыкания со временем работы $T(n)$ вытекает алгоритм умножения булевых матриц за время $O(T(n))$.

28.2.5

Применим ли основанный на теореме 28.2 алгоритм обращения матриц к матрицам над полем целых чисел по модулю 2? Поясните свой ответ.

28.2.6 ★

Обобщите алгоритм обращения матриц из теоремы 28.2 для матриц с комплексными числами и докажите корректность вашего обобщения. (*Указание:* вместо транспонирования матрицы A воспользуйтесь ***сопряженно-транспонированной*** (conjugate transpose) матрицей A^*, которая получается из исходной путем транспонирования и замены всех элементов комплексно сопряженными числами. Вместо симметричных матриц рассмотрите ***эрмитовы*** (Hermitian) матрицы, обладающие тем свойством, что $A = A^*$.)

28.3. Симметричные положительно определенные матрицы и метод наименьших квадратов

Симметричные положительно определенные матрицы обладают рядом интересных и полезных свойств. Например, они невырождены и их LU-разложение можно выполнить, не опасаясь столкнуться с делением на 0. В этом разделе мы докажем ряд других важных свойств симметричных положительно определенных матриц и покажем их интересное применение — для получения приближений методом наименьших квадратов.

Первое свойство, которое мы докажем, пожалуй, наиболее фундаментальное.

Лемма 28.3

Любая симметричная положительно определенная матрица является невырожденной.

Доказательство. Предположим, что матрица A вырождена. Тогда согласно следствию Г.3 имеется такой ненулевой вектор x, что $Ax = 0$. Следовательно, $x^{\mathrm{T}}Ax = 0$, и A не может быть положительно определенной. ■

Доказательство того факта, что LU-разложение симметричных положительно определенных матриц можно выполнить, не опасаясь столкнуться с делением на 0, более сложное. Начнем с доказательства свойств некоторых определенных подматриц A. Определим k-ю ***главную подматрицу*** (leading submatrix) A как матрицу A_k, состоящую из пересечения k первых строк и k первых столбцов A.

Лемма 28.4

Если A представляет собой симметричную положительно определенную матрицу, то все ее главные подматрицы симметричные и положительно определенные.

Доказательство. То, что каждая главная подматрица A_k является симметричной, очевидно. Для доказательства того, что она положительно определенная, вос-

пользуемся методом от противного. Если A_k не является положительно определенной, то существует вектор $x_k \neq 0$ размером k, такой, что $x_k^{\mathrm{T}} A_k x_k \leq 0$. Пусть матрица A имеет размер $n \times n$ и

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \tag{28.14}$$

с подматрицами B (размером $(n-k) \times k$) и C (размером $(n-k) \times (n-k)$). Определим вектор $x = \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix}^{\mathrm{T}}$ размером n, в котором после x_k следуют $n-k$ нулей. Тогда

$$\begin{aligned} x^{\mathrm{T}} A x &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^{\mathrm{T}} A_k x_k \\ &\leq 0 , \end{aligned}$$

что противоречит условию, что матрица A положительно определенная. ■

Теперь обратимся к некоторым важным свойствам дополнения Шура. Пусть A является симметричной положительно определенной матрицей, а A_k — главной подматрицей A размером $k \times k$. Вновь разделим A на части, как в (28.14). Обобщим определение (28.9) для определения ***дополнения Шура*** матрицы A относительно подматрицы A_k (Schur complement of A with respect to A_k) как

$$S = C - B A_k^{-1} B^{\mathrm{T}} . \tag{28.15}$$

(Согласно лемме 28.4 A_k — симметричная положительно определенная матрица; следовательно, в соответствии с леммой 28.3 A_k^{-1} существует, и дополнение Шура S вполне определено.) Заметим, что прежнее определение (28.9) дополнения Шура согласуется с определением (28.15), если положить $k = 1$.

Следующая лемма показывает, что дополнение Шура симметричной положительно определенной матрицы является симметричным положительно определенным. Этот результат используется в теореме 28.2, а следствие из него необходимо для доказательства корректности LU-разложения симметричных положительно определенных матриц.

Лемма 28.5 (Лемма о дополнении Шура)

Если A представляет собой симметричную положительно определенную матрицу, а A_k — главная подматрица A размером $k \times k$, то дополнение Шура S матрицы A относительно подматрицы A_k является симметричным положительно определенным.

Доказательство. Поскольку матрица A симметрична, симметрична также подматрица C. Согласно упр. Г.2.6 произведение $BA_k^{-1}B^{\mathrm{T}}$ симметричное, так что в соответствии с упр. Г.1.1 дополнение Шура S также является симметричным.

Остается показать, что дополнение Шура S положительно определенное. Рассмотрим разделение A (28.14). Для любого ненулевого вектора x в соответствии с предположением о том, что A является положительно определенной матрицей, выполняется соотношение $x^{\mathrm{T}}Ax > 0$. Разобьем x на два подвектора, y и z, совместимые с A_k и C соответственно. В силу существования A_k^{-1} имеем

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}}z \\ By + Cz \end{pmatrix} \\ &= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\ &= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z\,. \end{aligned} \tag{28.16}$$

(Корректность приведенного выражения можно проверить непосредственным вычислением.) Последнее равенство соответствует выделению полного квадрата из квадратичной формы (см. упр. 28.3.2).

Поскольку неравенство $x^{\mathrm{T}}Ax > 0$ выполняется для любого ненулевого вектора x, выберем произвольный ненулевой z и выберем $y = -A_k^{-1}B^{\mathrm{T}}z$, что удаляет первое слагаемое в (28.16), оставляя только

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

в качестве значения всего выражения. Итак, для любого $z \neq 0$ мы имеем $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, так что дополнение Шура S является положительно определенным. ■

Следствие 28.6

LU-разложение симметричной положительно определенной матрицы никогда не приводит к делению на 0.

Доказательство. Пусть A представляет собой симметричную положительно определенную матрицу. Докажем несколько более строгое утверждение, чем формулировка данного следствия: каждый ведущий элемент строго положителен. Первым ведущим элементом является a_{11}. Этот элемент положителен по определению положительно определенной матрицы A, так как его можно получить с помощью единичного вектора e_1 следующим образом: $a_{11} = e_1^{\mathrm{T}}Ae_1 > 0$. Поскольку на следующем шаге рекурсии LU-разложение применяется к дополнению Шура матрицы A относительно подматрицы $A_1 = (a_{11})$, из леммы 28.5 по индукции следует, что все ведущие элементы положительны. ■

Метод наименьших квадратов

Одним из важных приложений симметричных положительно определенных матриц является подбор кривой для заданного множества экспериментальных то-

чек. Предположим, что дано множество из m точек

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) ,$$

где значения y_i содержат ошибки измерений. Нужно найти функцию $F(x)$, такую, что ошибки аппроксимации

$$\eta_i = F(x_i) - y_i \tag{28.17}$$

малы при $i = 1, 2, \ldots, m$. Вид функции F зависит от рассматриваемой задачи, и здесь мы будем считать, что она имеет вид линейной взвешенной суммы

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) ,$$

где количество слагаемых n и набор ***базисных функций*** (basis functions) f_j выбираются на основе знаний о рассматриваемой задаче. Зачастую в качестве базисных функций выбираются $f_j(x) = x^{j-1}$, т.е. функция F представляет собой полином степени $n-1$ от x:

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1} .$$

Таким образом, для заданных m экспериментальных точек $(x_1, y_1), (x_2, y_2), \ldots,$ (x_m, y_m) необходимо вычислить n коэффициентов $c_1, c_2, \ldots, c_n$, минимизирующих ошибки приближения $\eta_1, \eta_2, \ldots, \eta_m$.

Выбрав $n = m$, можно *точно* вычислить все y_i в (28.17). Выбор такой функции F с высокой степенью не слишком удачен, так как, помимо данных, он учитывает и весь "шум", что приводит к плохим результатам при использовании F для предсказания значения y для некоторого x, измерения для которого еще не выполнялись. Обычно гораздо лучшие результаты получаются при n, значительно меньшем, чем m, поскольку тогда есть надежда выбрать такие коэффициенты c_j, которые позволят отфильтровать шум, вносимый ошибками измерений. Для выбора значения n имеются определенные теоретические предпосылки, но данная тема лежит за пределами нашей книги. В любом случае, когда выбрано n, меньшее, чем m, мы получаем переопределенную систему линейных уравнений, приближенное решение которой хотим найти. Покажем, каким образом это можно сделать.

Пусть

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \ldots & f_n(x_m) \end{pmatrix}$$

обозначает матрицу значений базисных функций в заданных точках, т.е. $a_{ij} = f_j(x_i)$, и пусть $c = (c_k)$ обозначает искомый вектор коэффициентов размером n.

Тогда

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

представляет собой вектор размером m "предсказанных значений" y, а вектор

$$\eta = Ac - y$$

является вектором ***невязок*** (ошибок приближения — approximation error) размером m.

Для минимизации невязок будем минимизировать норму вектора ошибок η, что отражено в названии "решение методом ***наименьших квадратов***" (least-square solution), так как

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2\right)^{1/2}$$

Поскольку

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)^2 ,$$

можно минимизировать $\|\eta\|$, дифференцировав $\|\eta\|^2$ по всем c_k и приравняв полученные производные к 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right) a_{ik} = 0 . \qquad (28.18)$$

n уравнений (28.18) с $k = 1, 2, \dots, n$ эквивалентны одному матричному уравнению

$$(Ac - y)^{\mathrm{T}} A = 0 ,$$

или, что то же самое (см. упр. Г.1.2),

$$A^{\mathrm{T}}(Ac - y) = 0 ,$$

откуда вытекает

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y . \qquad (28.19)$$

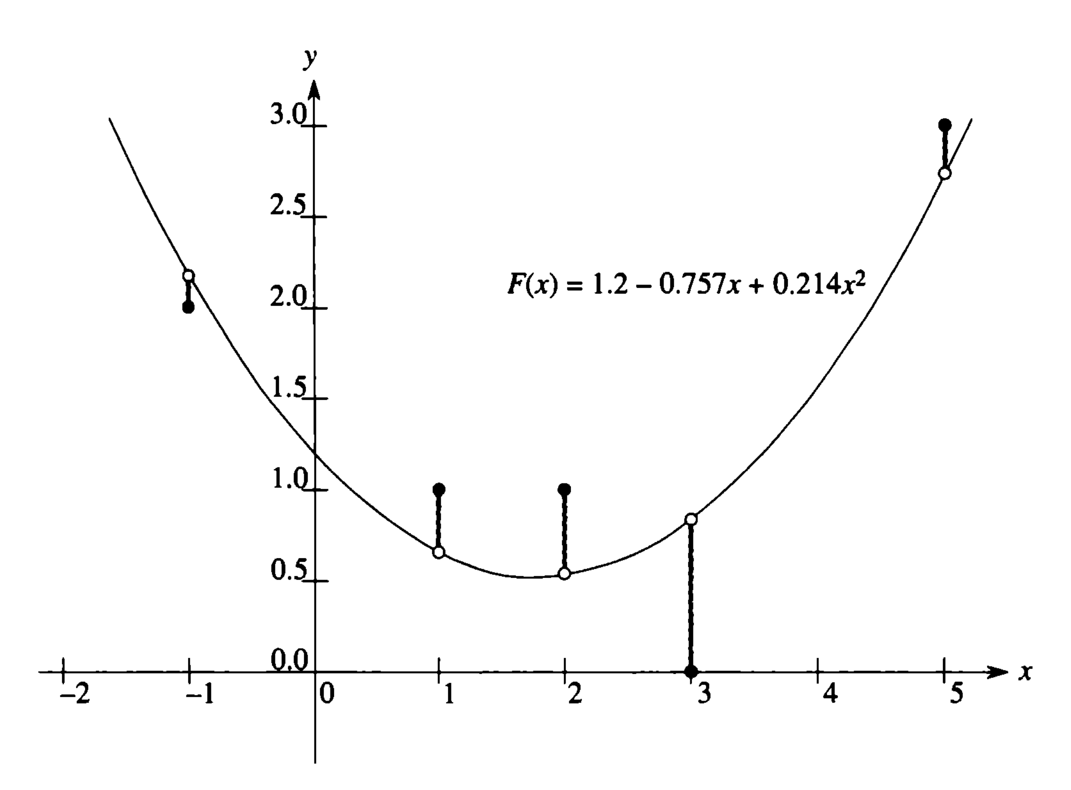


Рис. 28.3. Применение метода наименьших квадратов для получения приближения пяти экспериментальных точек $\{(-1,2),(1,1),(2,1),(3,0),(5,3)\}$ квадратичным полиномом. Черным цветом показаны экспериментальные точки, а белым — значения, предсказываемые квадратичным полиномом $F(x) = 1.2 - 0.757x + 0.214x^2$, минимизирующим сумму квадратов невязок. Серыми линиями указаны невязки для каждой точки данных.

В математической статистике такое уравнение называется ***нормальным уравнением*** (normal equation). Согласно упр. Г.1.2 матрица $A^{\mathrm{T}}A$ симметрична, и если A имеет полный столбцовый ранг, то по теореме Г.6 матрица $A^{\mathrm{T}}A$ положительно определенная. Следовательно, существует обратная матрица $(A^{\mathrm{T}}A)^{-1}$, и решение уравнения (28.19) имеет вид

$$\begin{aligned} c &= \left((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\right)y \\ &= A^{+}y \ , \end{aligned} \tag{28.20}$$

где матрица $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ является ***псевдообратной*** (pseudoinverse) к матрице A. Псевдообратная матрица является естественным обобщением обратной матрицы для случая, когда исходная матрица не является квадратной (сравните уравнение (28.20), дающее приближенное решение уравнения $Ac = y$, с точным решением $A^{-1}b$ уравнения $Ax = b$).

В качестве примера рассмотрим пять экспериментальных точек,

$$\begin{aligned} (x_1, y_1) &= (-1, 2) \ , \\ (x_2, y_2) &= (1, 1) \ , \\ (x_3, y_3) &= (2, 1) \ , \\ (x_4, y_4) &= (3, 0) \ , \\ (x_5, y_5) &= (5, 3) \ , \end{aligned}$$

которые показаны на рис. 28.3 черным цветом. Необходимо найти приближение экспериментальных данных квадратичным полиномом

$$F(x) = c_1 + c_2x + c_3x^2 .$$

Начнем с матрицы значений базисных функций:

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} .$$

Ее псевдообратная матрица имеет вид

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix} .$$

Умножив y на A^+, получаем вектор коэффициентов

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix} ,$$

соответствующий квадратичному полиному

$$F(x) = 1.200 - 0.757x + 0.214x^2 ,$$

который представляет собой наилучшее квадратичное приближение экспериментальных данных в смысле наименьших квадратов.

На практике нормальное уравнение (28.19) решается путем умножения y на A^{T} с последующим поиском LU-разложения $A^{\mathrm{T}}A$. Если матрица A имеет полный ранг, то матрица $A^{\mathrm{T}}A$ гарантированно невырожденная, поскольку она симметричная и положительно определенная (см. упр. Г.1.2 и теорему Г.6).

Упражнения

28.3.1

Докажите, что все диагональные элементы симметричной положительно определенной матрицы положительны.

28.3.2

Пусть $A = \left(\begin{smallmatrix} a & b \\ b & c \end{smallmatrix}\right)$ является симметричной положительно определенной матрицей размером 2×2. Докажите, что ее детерминант $ac - b^2$ положителен, выделив полный квадрат так, как это было сделано в доказательстве леммы 28.5.

28.3.3

Докажите, что максимальный элемент симметричной положительно определенной матрицы находится на ее диагонали.

28.3.4

Докажите, что детерминант каждой главной подматрицы симметричной положительно определенной матрицы положителен.

28.3.5

Обозначим через A_k k-ю главную подматрицу симметричной положительно определенной матрицы A. Докажите, что k-й ведущий элемент при LU-разложении равен $\det(A_k)/\det(A_{k-1})$ (полагаем, что $\det(A_0) = 1$).

28.3.6

Найдите функцию вида

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x ,$$

являющуюся наилучшим приближением в смысле наименьших квадратов экспериментальных данных

$$(1,1), (2,1), (3,3), (4,8) .$$

28.3.7

Покажите, что псевдообратная матрица A^+ удовлетворяет четырем следующим уравнениям:

$$\begin{aligned} AA^+A &= A , \\ A^+AA^+ &= A^+ , \\ (AA^+)^{\mathrm{T}} &= AA^+ , \\ (A^+A)^{\mathrm{T}} &= A^+A . \end{aligned}$$

Задачи

28.1. Трехдиагональные системы линейных уравнений

Рассмотрим трехдиагональную матрицу

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} .$$

а. Найдите LU-разложение A.

б. Решите уравнение $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ с применением прямой и обратной подстановок.

в. Найдите матрицу, обратную матрице A.

г. Покажите, как для любой симметричной положительно определенной трехдиагональной матрицы A размером $n \times n$ и произвольного вектора b размером n можно решить уравнение $Ax = b$ с помощью LU-разложения за время $O(n)$. Докажите, что любой другой алгоритм, основанный на обращении матрицы A, имеет асимптотически большее время работы в худшем случае.

д. Покажите, как использование LUP-разложения позволяет решить уравнение $Ax = b$, где A — невырожденная трехдиагональная матрица размером $n \times n$, а b — произвольный вектор размером n, за время $O(n)$.

28.2. Сплайны

Один из практических методов интерполяции набора точек с помощью кривых заключается в использовании ***кубических сплайнов*** (cubic splines). Пусть задано множество $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ из $n+1$ пары "точка–значение", причем $x_0 < x_1 < \cdots < x_n$. Необходимо провести через эти точки кусочно-кубическую кривую (сплайн) $f(x)$. Кривая $f(x)$ состоит из n кубических полиномов $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ $(i = 0, 1, \ldots, n-1)$. Когда x находится в диапазоне $x_i \leq x \leq x_{i+1}$, значение кривой вычисляется как $f(x) = f_i(x - x_i)$. Точки x_i, в которых "состыковываются" кубические полиномы, называются ***узлами*** (knots). Для простоты будем считать, что $x_i = i$ для $i = 0, 1, \ldots, n$.

Чтобы обеспечить непрерывность $f(x)$, потребуем выполнения условий

$$f(x_i) = f_i(0) = y_i\ ,$$
$$f(x_{i+1}) = f_i(1) = y_{i+1}$$

для $i = 0, 1, \ldots, n-1$. Чтобы функция $f(x)$ была достаточно гладкой, мы также потребуем непрерывности первой производной в каждом узле:

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

для $i = 0, 1, \ldots, n-2$.

а. Предположим, что нам даны не только пары $\{(x_i, y_i)\}$, но и значения первых производных $D_i = f'(x_i)$ в каждом узле $i = 0, 1, \ldots, n$. Выразите коэффициенты a_i, b_i, c_i и d_i через y_i, y_{i+1}, D_i и D_{i+1} (напомним, что $x_i = i$). Сколько времени потребуется для вычисления $4n$ коэффициентов при таких условиях?

Остается вопрос о вычислении первой производной $f(x)$ в узлах. Один из методов их определения состоит в том, чтобы обеспечить в узлах непрерывность второй производной

$$f''(x_{i+1}) = f''_i(1) = f''_{i+1}(0)$$

для $i = 0, 1, \ldots, n-2$. Мы полагаем, что в первом и последнем узлах $f''(x_0) = f_0''(0) = 0$ и $f''(x_n) = f_{n-1}''(1) = 0$. Это предположение делает $f(x)$ ***естественным*** кубическим сплайном (natural cubic spline).

б. Используя непрерывность второй производной, покажите, что для $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) . \quad (28.21)$$

в. Покажите, что

$$2D_0 + D_1 = 3(y_1 - y_0) , \quad (28.22)$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) . \quad (28.23)$$

г. Перепишите уравнения (28.21)–(28.23) в матричном виде с использованием вектора неизвестных $D = \langle D_0, D_1, \ldots, D_n \rangle$. Какими свойствами обладает матрица вашего уравнения?

д. Докажите, что множество из $n+1$ пар "точка–значение" может быть интерполировано с помощью естественного кубического сплайна за время $O(n)$ (см. задачу 28.1).

е. Покажите, как построить естественный кубический сплайн, который интерполирует множество из $n+1$ точек (x_i, y_i), где $x_0 < x_1 < \cdots < x_n$ и где значения x_i не обязательно равны i. Какое матричное уравнение для этого нужно решить и сколько времени для этого потребуется?

Заключительные замечания

Имеется множество превосходных работ, в которых вопросы числовых и научных вычислений рассматриваются более детально, чем в данной книге. В особенности порекомендуем следующие книги: Джордж (George) и Лью (Liu) [131], Голуб (Golub) и ван Лоан (van Loan) [143], Пресс (Press), Тукольски (Teukolsky), Веттерлинг (Vetterling) и Фланнери (Flannery) [281,282], Стренг (Strang) [321,322].

В книге Голуба и ван Лоана [143] рассматриваются вопросы численной устойчивости. Авторы показали, почему $\det(A)$ не может служить хорошим показателем устойчивости матрицы A, предложив вместо него использовать величину $\|A\|_\infty \|A^{-1}\|_\infty$, где $\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. Кроме того, рассмотрен вопрос вычисления указанного значения без реального вычисления обратной матрицы A^{-1}.

Метод исключения неизвестных Гаусса, на котором основаны алгоритмы LU- и LUP-разложения, был первым систематическим методом решения систем линейных уравнений и одним из первых численных алгоритмов. Хотя он был известен и ранее, его открытие обычно приписывают К. Ф. Гауссу (C. F. Gauss) (1777–1855). В своей знаменитой работе [323] Штрассен показал, что матрица размером $n \times n$ может быть обращена за время $O(n^{\lg 7})$. Виноград (Winograd) [356]

доказал, что умножение матриц не сложнее обращения, а обратная оценка была получена Ахо (Aho), Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5].

Еще одним важным разложением матриц является ***сингулярное разложение*** (singular value decomposition — SVD). SVD-разложение матрицы A размером $m \times n$ представляет собой разложение $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, где Σ — матрица размером $m \times n$, в которой ненулевые элементы находятся только на диагонали, Q_1 — матрица размером $m \times m$ со взаимно ортонормальными столбцами, а матрица Q_2 имеет размер $n \times n$ и также обладает свойством взаимной ортонормальности столбцов. Два вектора называются ***ортонормальными*** (orthonormal), если их скалярное произведение равно 0, а норма каждого вектора равна 1. Сингулярное разложение хорошо изложено в книгах Стренга [321, 322], а также Голуба и ван Лоана [143].

Прекрасное изложение теории симметричных положительно определенных матриц (и линейной алгебры вообще) содержится в книге Стренга [322].

Глава 29. Линейное программирование

Многие задачи можно сформулировать как задачи максимизации или минимизации некой цели в условиях ограниченности ресурсов и наличия конкурирующих ограничений. Если удастся задать цель в виде линейной функции нескольких переменных и сформулировать ограничения в виде равенств или неравенств, связывающих эти переменные, можно получить ***задачу линейного программирования*** (linear-programming problem). Задачи линейного программирования часто встречаются в разнообразных практических приложениях. Начнем их изучение на примере подготовки предвыборной кампании.

Политическая задача

Представьте себя на месте политика, стремящегося выиграть выборы. В вашем избирательном округе есть районы трех типов — городские, пригородные и сельские. В этих районах зарегистрировано соответственно 100, 200 и 50 тысяч избирателей. Чтобы добиться успеха, желательно получить большинство голосов в каждом из трех регионов. Вы честный человек и никогда не будете давать обещания, в которые сами не верите. Однако вам известно, что определенные пункты программы могут помочь завоевать голоса тех или иных групп избирателей. Основными пунктами программы являются строительство дорог, контроль над использованием огнестрельного оружия, сельскохозяйственные субсидии и налог на бензин, направляемый на улучшение работы общественного транспорта. Согласно исследованиям вашего предвыборного штаба можно оценить, сколько голосов будет приобретено или потеряно в каждой группе избирателей, если потратить тысячу долларов на пропаганду по каждому пункту программы. Эта информация представлена в таблице на рис. 29.1. Каждый элемент данной таблицы показывает, сколько тысяч избирателей из городских, пригородных и сельских районов удастся привлечь, потратив тысячу долларов на агитацию в поддержку определенного пункта предвыборной программы. Отрицательные элементы отражают потерю голосов. Задача состоит в минимизации суммы, которая позволит завоевать 50 тысяч голосов горожан, 100 тысяч голосов избирателей из пригородных зон и 25 тысяч голосов сельских жителей.

Методом проб и ошибок можно выработать стратегию, которая позволит получить необходимое количество голосов, но затраты на такую стратегию могут оказаться не самыми низкими. Например, можно выделить 20 тысяч долларов на пропаганду строительства дорог, 0 долларов на агитацию в пользу контроля над

Пункт программы	Горожане	Жители пригорода	Сельские жители
Строительство дорог	−2	5	3
Контроль над оружием	8	2	−5
Сельскохозяйственные субсидии	0	0	10
Налог на бензин	10	0	−2

Рис. 29.1. Влияние предвыборной политики на настроения избирателей. Каждая запись представляет собой количество тысяч голосов горожан, жителей пригорода и сельских жителей, получаемых при затрате тысячи долларов на рекламу определенных пунктов предвыборной программы. Отрицательные элементы отражают потерю голосов.

использованием оружия, 4 тысячи долларов на пропаганду сельскохозяйственных субсидий и 9 тысяч долларов на агитацию за введение налога на бензин. При этом удастся привлечь $20(-2) + 0(8) + 4(0) + 9(10) = 50$ тысяч голосов горожан, $20(5) + 0(2) + 4(0) + 9(0) = 100$ тысяч голосов жителей пригородов и $20(3) + 0(-5) + 4(10) + 9(-2) = 82$ тысячи голосов сельских жителей. Таким образом, будет получено ровно необходимое количество голосов в городских и пригородных районах и большее количество голосов в сельской местности. (Получается, что в сельской местности привлечено голосов больше, чем имеется избирателей!) Чтобы получить эти голоса, придется потратить на пропаганду $20 + 0 + 4 + 9 = 33$ тысячи долларов.

Возникает естественный вопрос: является ли данная стратегия наилучшей из возможных, т.е. можно ли достичь поставленных целей, потратив на пропаганду меньше денег? Ответ на этот вопрос можно получить, продолжая действовать методом проб и ошибок, однако вместо этого хотелось бы иметь некий систематический метод для ответа на подобные вопросы. Сформулируем данный вопрос математически. Введем четыре переменные:

- x_1 — сумма (в тысячах долларов), потраченная на пропаганду программы строительства дорог;
- x_2 — сумма (в тысячах долларов), потраченная на агитацию в пользу контроля над использованием оружия;
- x_3 — сумма (в тысячах долларов), потраченная на пропаганду программы сельскохозяйственных субсидий;
- x_4 — сумма (в тысячах долларов), потраченная на агитацию за введение налога на бензин.

Требование получить не менее 50 тысяч голосов избирателей-горожан можно записать в виде неравенства

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 . \tag{29.1}$$

Аналогично требования получить не менее 100 тысяч голосов избирателей, живущих в пригороде, и 25 тысяч голосов избирателей в сельской местности можно

записать как неравенства

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

и

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25\ . \tag{29.3}$$

Любые значения переменных x_1, x_2, x_3, x_4, удовлетворяющие неравенствам (29.1)–(29.3), позволят получить достаточное количество голосов избирателей в каждом регионе. Чтобы сделать затраты минимально возможными, необходимо минимизировать сумму, затраченную на пропаганду, т.е. минимизировать выражение

$$x_1 + x_2 + x_3 + x_4\ . \tag{29.4}$$

Хотя отрицательная агитация часто встречается в политической борьбе, отрицательных затрат на пропаганду не бывает. Поэтому следует записать условия

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0 \ \text{ и } \ x_4 \geq 0\ . \tag{29.5}$$

Объединив неравенства (29.1)–(29.3) и (29.5) для минимизации (29.4), мы получаем то, что известно как задача линейного программирования. Запишем ее следующим образом:

минимизировать
$$x_1 + x_2 + x_3 + x_4 \tag{29.6}$$

при условиях

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geq 0. \tag{29.10}$$

Решение этой задачи линейного программирования позволит политику получить оптимальную стратегию предвыборной агитации.

Общий вид задач линейного программирования

В общем случае в задаче линейного программирования требуется оптимизировать некую линейную функцию при условии выполнения множества линейных неравенств. Для данных действительных чисел $a_1, a_2, \ldots, a_n$ и множества переменных $x_1, x_2, \ldots, x_n$ ***линейная функция*** этих переменных f определяется как

$$f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j\ .$$

Если b представляет собой действительное число, а f является линейной функцией, то уравнение

$$f(x_1, x_2, \ldots, x_n) = b$$

называется ***линейным равенством***, а неравенства

$$f(x_1, x_2, \ldots, x_n) \leq b$$

и

$$f(x_1, x_2, \dots, x_n) \geq b$$

называются линейными неравенствами. Термин ***линейные ограничения*** применяется как к линейным равенствам, так и к линейным неравенствам. В линейном программировании не допускается использование строгих неравенств. Формально ***задача линейного программирования*** является задачей минимизации или максимизации линейной функции при соблюдении конечного множества линейных ограничений. Если выполняется минимизация, то такая задача называется ***задачей минимизации***, а если выполняется максимизация, то такая задача называется ***задачей максимизации***.

Вся оставшаяся часть данной главы будет посвящена формулированию и решению задач линейного программирования. Для решения задач линейного программирования существует несколько алгоритмов с полиномиальным временем выполнения, однако изучать их в данной главе мы не будем. Вместо этого мы рассмотрим самый старый алгоритм линейного программирования — симплекс-алгоритм. В наихудшем случае симплекс-алгоритм не выполняется за полиномиальное время, однако обычно он достаточно эффективен и широко используется на практике.

Обзор задач линейного программирования

Чтобы описывать свойства задач линейного программирования и алгоритмы их решения, удобно договориться, в каких формах их записывать. В данной главе мы будем использовать две формы: ***стандартную*** и ***каноническую*** (slack). Их строгое определение будет дано в разделе 29.1. Неформально стандартная форма задачи линейного программирования представляет собой задачу максимизации линейной функции при соблюдении линейных *неравенств*, в то время как каноническая форма является задачей максимизации линейной функции при соблюдении линейных *равенств*. Обычно мы будем использовать стандартную форму задач линейного программирования, однако при описании принципа работы симплекс-алгоритма удобнее использовать каноническую форму. На данном этапе ограничимся рассмотрением задачи максимизации линейной функции от n переменных при условии выполнения m линейных неравенств.

Начнем с рассмотрения следующей задачи линейного программирования с двумя переменными:

максимизировать $\quad x_1 + x_2 \qquad (29.11)$

при условиях

$$4x_1 - x_2 \leq 8 \qquad (29.12)$$

$$2x_1 + x_2 \leq 10 \qquad (29.13)$$

$$5x_1 - 2x_2 \geq -2 \qquad (29.14)$$

$$x_1, x_2 \geq 0. \qquad (29.15)$$

Любой набор значений переменных x_1 и x_2, удовлетворяющий всем ограничениям (29.12)–(29.15), называется ***допустимым решением*** (feasible solution) дан-

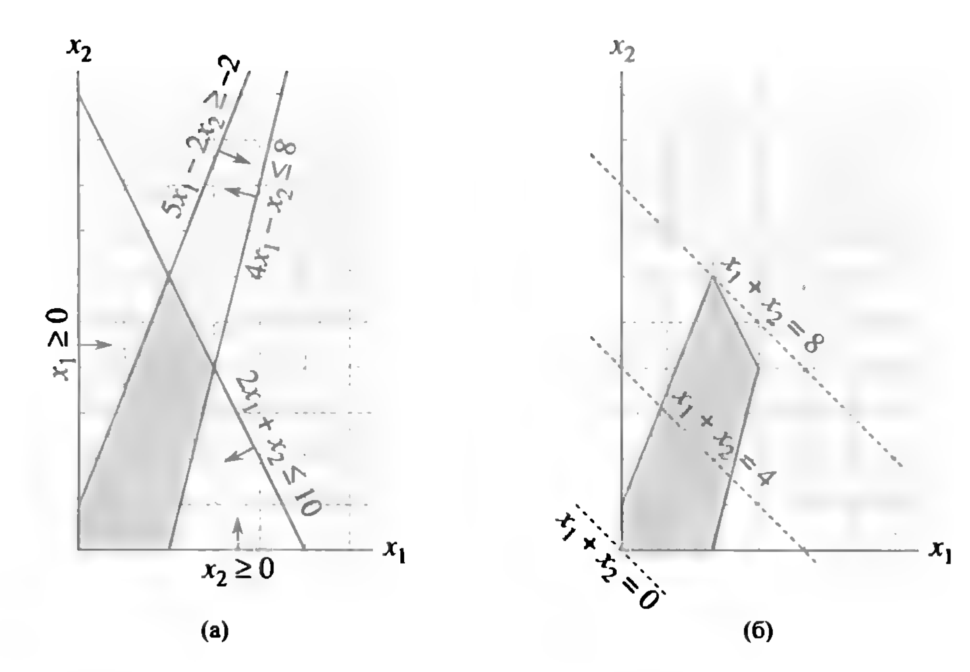


Рис. 29.2. **(а)** Задача линейного программирования (29.12)–(29.15). Каждое ограничение представлено линией и направлением. Пересечение ограничений, представляющее область допустимых решений, заштриховано. **(б)** Пунктирными линиями показаны точки, для которых целевое значение равно соответственно 0, 4 и 8. Оптимальным решением данной задачи линейного программирования является $x_1 = 2$ и $x_2 = 6$ с целевым значением 8.

ной задачи линейного программирования. Если изобразить эти ограничения в декартовой системе координат (x_1, x_2), как показано на рис. 29.2, (а), то множество допустимых решений (заштрихованная область на рисунке) образует выпуклую область[1] в двумерном пространстве. Эта выпуклая область называется ***областью допустимых решений***, или допустимой областью (feasible region). Функция, которую мы хотим максимизировать, называется ***целевой функцией*** (objective function). Теоретически можно было бы оценить значение целевой функции $x_1 + x_2$ в каждой точке допустимой области (значение целевой функции в определенной точке называется ***целевым значением*** (objective value)). Затем можно найти точку, в которой целевое значение максимально; она и будет оптимальным решением. В данном примере (как и в большинстве задач линейного программирования) допустимая область содержит бесконечное множество точек, поэтому хотелось бы найти способ, который позволит находить точку с максимальным целевым значением, не прибегая к вычислению значений целевой функции в каждой точке допустимой области.

В двумерном случае оптимальное решение можно найти с помощью графической процедуры. Множество точек, для которых $x_1 + x_2 = z$, при любом z представляет собой прямую с коэффициентом наклона -1. Если мы построим график функции $x_1 + x_2 = 0$, получится прямая с коэффициентом наклона -1, проходящая через начало координат, как показано на рис. 29.2, (б). Пересечение

---

[1]Интуитивно выпуклая область определяется как область, удовлетворяющая тому требованию, что для любых двух точек, принадлежащих области, все точки соединяющего их отрезка также должны принадлежать этой области.

данной прямой и допустимой области представляет собой множество допустимых решений, целевое значение в которых равно 0. В данном случае пересечением прямой и допустимой областей является точка $(0, 0)$. В общем случае для любого z пересечением прямой $x_1 + x_2 = z$ с допустимой областью является множество допустимых решений, в которых целевое значение равно z. На рис. 29.2, (б) показаны прямые $x_1 + x_2 = 0$, $x_1 + x_2 = 4$ и $x_1 + x_2 = 8$. Поскольку допустимая область на рис. 29.2 ограничена, должно существовать некоторое максимальное значение z, для которого пересечение прямой $x_1 + x_2 = z$ и допустимой области является непустым множеством. Любая точка этого пересечения является оптимальным решением задачи линейного программирования; в данном случае такой точкой является $x_1 = 2$, $x_2 = 6$ с целевым значением 8.

То, что оптимальное решение задачи линейного программирования оказалось в некоторой вершине допустимой области, не случайно. Максимальное значение z, при котором прямая $x_1 + x_2 = z$ пересекает допустимую область, должно находиться на границе допустимой области, поэтому пересечение данной прямой и границы допустимой области может быть либо вершиной, либо отрезком. Если пересечение является вершиной, то существует единственное оптимальное решение, находящееся в данной вершине. Если же пересечение является отрезком, то все точки этого отрезка имеют одинаковое целевое значение и являются оптимальными решениями; в частности, оптимальными решениями являются оба конца отрезка. Каждый конец отрезка является вершиной, поэтому в данном случае также существует оптимальное решение в вершине допустимой области.

Несмотря на то что для задач линейного программирования, в которых число переменных больше двух, простое графическое решение построить невозможно, наши интуитивные соображения остаются в силе. В случае трех переменных каждое ограничение описывается полупространством в трехмерном пространстве. Пересечение этих полупространств образует допустимую область. Множество точек, в которых целевая функция имеет значение z, теперь представляет собой некоторую плоскость. Если все коэффициенты целевой функции неотрицательны и начало координат является допустимым решением рассматриваемой задачи линейного программирования, то при движении этой плоскости по направлению от начала координат получаются точки с возрастающими значениями целевой функции. (Если начало координат не является допустимым решением или некоторые коэффициенты целевой функции отрицательны, интуитивная картина будет немного сложнее.) Как и в двумерном случае, поскольку допустимая область выпукла, множество точек, в которых достигается оптимальное целевое значение, должно содержать вершину допустимой области. Аналогично в случае n переменных каждое ограничение определяет полупространство в n-мерном пространстве. Допустимая область, образуемая пересечением этих полупространств, называется ***симплексом*** (simplex). Целевая функция в этом случае представляет собой гиперплоскость, и благодаря выпуклости допустимой области оптимальное решение находится в некоторой вершине симплекса.

Симплекс-алгоритм получает на вход задачу линейного программирования и возвращает оптимальное решение. Он начинает работу в некоторой вершине симплекса и выполняет последовательность итераций. В каждой итерации осу-

ществляется переход вдоль ребра симплекса из текущей вершины в соседнюю, целевое значение в которой не меньше (а обычно больше), чем в текущей вершине. Симплекс-алгоритм завершается при достижении локального максимума, т.е. вершины, все соседние вершины которой имеют меньшее целевое значение. Поскольку допустимая область является выпуклой, а целевая функция линейна, локальный оптимум в действительности является глобальным. В разделе 29.4 мы воспользуемся понятием двойственности, чтобы показать, что решение, полученное с помощью симплекс-алгоритма, действительно оптимально.

Хотя геометрическое представление позволяет наглядно проиллюстрировать операции симплекс-алгоритма, мы не будем непосредственно обращаться к нему при подробном рассмотрении симплекс-метода в разделе 29.3. Вместо этого воспользуемся алгебраическим представлением. Сначала запишем задачу линейного программирования в канонической форме в виде набора линейных равенств. Эти линейные равенства выражают одни переменные, называемые *базисными*, через другие переменные, называемые *небазисными*. Переход от одной вершины к другой осуществляется путем замены одной из базисных переменных небазисной переменной. Данная операция называется *замещением* и алгебраически заключается в переписывании задачи линейного программирования в эквивалентной канонической форме.

Приведенный выше пример с двумя переменными был исключительно простым. В данной главе нам предстоит рассмотреть и более сложные случаи: задачи линейного программирования, не имеющие решений; задачи, не имеющие конечного оптимального решения, и задачи, для которых начало координат не является допустимым решением.

Приложения линейного программирования

Линейное программирование имеет широкий спектр приложений. В любом учебнике по исследованию операций содержится множество примеров задач линейного программирования; линейное программирование становится стандартным инструментом, который преподается студентам большинства школ бизнеса. Разработка сценария предвыборной борьбы — лишь один типичный пример. Приведем еще два примера успешного использования линейного программирования.

- Авиакомпания составляет график работы экипажей. Федеральное авиационное агентство установило ряд ограничений, таких как ограничение времени непрерывной работы для каждого члена экипажа и требование, чтобы каждый конкретный экипаж работал только на самолетах одной модели на протяжении одного месяца. Авиакомпания планирует назначить экипажи на все рейсы, задействовав как можно меньше сотрудников (членов экипажей).
- Нефтяная компания выбирает место бурения скважины. С размещением буровой в каждом конкретном месте связаны определенные затраты и ожидаемая прибыль в виде некоторого количества баррелей добытой нефти, рассчитанная на основе проведенных геологических исследований. Средства, выделяемые на бурение новых скважин, ограничены, и компания хочет максимизировать ожидаемое количество добываемой нефти исходя из заданного бюджета.

Задачи линейного программирования полезны также при моделировании и решении задач теории графов и комбинаторных задач, таких как задачи, приведенные в данной книге. Мы уже встречались с частным случаем линейного программирования, который использовался для решения систем разностных ограничений в разделе 24.4. В разделе 29.2 мы покажем, как сформулировать в виде задач линейного программирования некоторые задачи теории графов и задачи транспортных сетей. В разделе 35.4 линейное программирование будет использоваться в качестве инструмента поиска приблизительного решения еще одной задачи теории графов.

Алгоритмы решения задач линейного программирования

В этой главе изучается симплекс-алгоритм. При аккуратном применении данный алгоритм на практике обычно позволяет быстро решать задачи линейного программирования общего вида. Однако при некоторых специально подобранных начальных значениях время выполнения симплекс-алгоритма может оказаться экспоненциальным. Первым алгоритмом с полиномиальным временем выполнения для решения задач линейного программирования был ***эллипсоидный алгоритм***, который на практике работает весьма медленно. Второй класс полиномиальных по времени алгоритмов содержит так называемые ***методы внутренней точки*** (interior-point methods). В отличие от симплекс-алгоритма, который движется по границе допустимой области и на каждой итерации рассматривает допустимое решение, являющееся вершиной симплекса, эти алгоритмы осуществляют движение по внутренней части допустимой области. Промежуточные решения являются допустимыми, но не обязательно находятся в вершинах симплекса, однако конечное решение является вершиной. Для задач большой размерности производительность алгоритмов внутренней точки может быть сравнима с производительностью симплекс-алгоритма, а иногда может и превышать ее. О том, где найти дополнительную информацию по указанным алгоритмам, говорится в заключительных замечаниях к данной главе.

Если ввести в задачу линейного программирования дополнительное требование, чтобы все переменные принимали целые значения, получится задача ***целочисленного линейного программирования***. В упр. 34.5.3 предлагается показать, что в этом случае даже задача поиска допустимого решения является NP-полной; поскольку ни для одной NP-полной задачи полиномиальные алгоритмы решения неизвестны, для задач целочисленного линейного программирования полиномиальные по времени алгоритмы также неизвестны. В отличие от них, задача линейного программирования общего вида может быть решена за полиномиальное время.

В данной главе для указания конкретного набора значений переменных в задаче линейного программирования с переменными $x = (x_1, x_2, \ldots, x_n)$ мы будем использовать обозначение $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$.

29.1. Стандартная и каноническая формы задачи линейного программирования

В данном разделе описываются стандартная и каноническая формы задач линейного программирования, которые будут полезны при формулировании задач и работе с ними. В стандартной форме все ограничения являются неравенствами, а в канонической форме все ограничения являются равенствами (за исключением ограничений, требующих, чтобы все переменные были неотрицательными).

Стандартная форма

В ***стандартной форме*** задаются n действительных чисел $c_1, c_2, \ldots, c_n$; m действительных чисел $b_1, b_2, \ldots, b_m$; и mn действительных чисел a_{ij}, где $i = 1, 2, \ldots, m$ и $j = 1, 2, \ldots, n$. Требуется найти n действительных чисел $x_1, x_2, \ldots, x_n$, которые

максимизируют $\sum_{j=1}^{n} c_j x_j$ (29.16)

при условиях

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ при } i = 1, 2, \ldots, m \quad (29.17)$$

$$x_j \geq 0 \text{ при } j = 1, 2, \ldots, n \, . \quad (29.18)$$

Обобщая введенную для двумерной задачи линейного программирования терминологию, будем называть выражение (29.16) ***целевой функцией***, а $n + m$ неравенств (29.17) и (29.18) — ***ограничениями***; n ограничений (29.18) называются ***ограничениями неотрицательности***. Произвольная задача линейного программирования не обязательно содержит ограничения неотрицательности, но в стандартной форме они необходимы. Иногда удобно записывать задачу линейного программирования в более компактной форме. Определим $m \times n$-матрицу $A = (a_{ij})$, m-мерный вектор $b = (b_i)$, n-мерный вектор $c = (c_j)$ и n-мерный вектор $x = (x_j)$. Тогда задачу линейного программирования (29.16)–(29.18) можно записать в следующем виде:

максимизировать $c^{\mathrm{T}} x$ (29.19)

при условиях

$$Ax \leq b \quad (29.20)$$

$$x \geq 0 \, . \quad (29.21)$$

В строке (29.19) $c^{\mathrm{T}} x$ представляет собой скалярное произведение двух векторов. В неравенстве (29.20) Ax является произведением матрицы и вектора, а $x \geq 0$ в (29.21) означает, что все компоненты вектора x должны быть неотрицательными. Таким образом, задачу линейного программирования в стандартной форме можно описать с помощью тройки (A, b, c), и мы примем соглашение, что A, b, и c всегда имеют указанную выше размерность.

Теперь введем терминологию для описания различных ситуаций, возникающих в линейном программировании. Некоторые термины уже использовались в двумерной задаче. Набор значений переменных $\bar{x}$, которые удовлетворяют всем ограничениям, называется ***допустимым решением***, в то время как набор значений переменных $\bar{x}$, не удовлетворяющий хотя бы одному ограничению, называется ***недопустимым решением***. Решению $\bar{x}$ соответствует ***целевое значение*** $c^{\mathrm{T}}\bar{x}$. Допустимое решение $\bar{x}$, целевое значение которого является максимальным среди всех допустимых решений, является ***оптимальным решением***, а его целевое значение $c^{\mathrm{T}}\bar{x}$ называется ***оптимальным целевым значением***. Если задача линейного программирования не имеет допустимых решений, она называется ***неразрешимой*** (infeasible), в противном случае она является ***разрешимой*** (feasible). Если задача линейного программирования имеет допустимые решения, но не имеет конечного оптимального целевого значения, она называется ***неограниченной*** (unbounded). В упр. 29.1.9 предлагается показать, что задача линейного программирования может иметь конечное оптимальное целевое значение даже в том случае, когда ее допустимая область неограничена.

Преобразование задач линейного программирования в стандартную форму

Любую задачу линейного программирования, в которой требуется минимизировать или максимизировать некую линейную функцию при наличии линейных ограничений, всегда можно преобразовать в стандартную форму. Исходная задача может находиться не в стандартной форме по четырем причинам.

1. Целевая функция минимизируется, а не максимизируется.
2. На некоторые переменные не наложены условия неотрицательности.
3. Некоторые ограничения имеют форму равенств, т.е. имеют знак равенства вместо знака “меньше или равно”.
4. Некоторые ограничения-неравенства вместо знака “меньше или равно” имеют знак “больше или равно”.

Преобразовывая задачу линейного программирования L в другую задачу линейного программирования L', мы бы хотели, чтобы оптимальное решение задачи L' позволяло найти оптимальное решение задачи L. Будем говорить, что две задачи максимизации, L и L', ***эквивалентны***, если для каждого допустимого решения $\bar{x}$ задачи L с целевым значением z существует соответствующее допустимое решение $\bar{x}'$ задачи L' с целевым значением z, а для каждого допустимого решения $\bar{x}'$ задачи L' с целевым значением z существует соответствующее допустимое решение $\bar{x}$ задачи L с целевым значением z. (Из этого определения не следует взаимно однозначное соответствие между допустимыми решениями.) Задача минимизации L и задача максимизации L' эквивалентны, если для каждого допустимого решения $\bar{x}$ задачи L с целевым значением z существует соответствующее допустимое решение $\bar{x}'$ задачи L' с целевым значением $-z$, а для каждого допустимого решения $\bar{x}'$ задачи L' с целевым значением z существует соответствующее допустимое решение $\bar{x}$ задачи L с целевым значением $-z$.

Теперь покажем, как поочередно избавиться от перечисленных выше возможных проблем. После устранения каждого несоответствия стандартной форме докажем, что новая задача линейного программирования эквивалентна старой.

Чтобы превратить задачу минимизации L в эквивалентную ей задачу максимизации L', достаточно просто изменить знаки коэффициентов целевой функции на противоположные. Поскольку L и L' имеют одинаковые множества допустимых решений и для любого допустимого решения целевое значение L противоположно целевому значению L', эти две задачи линейного программирования эквивалентны. Например, пусть исходная задача имеет вид

$$
\begin{array}{ll}
\text{минимизировать} & -2x_1 + 3x_2 \\
\text{при условиях} & \\
& \begin{array}{rcl} x_1 + \;\; x_2 & = & 7 \\ x_1 - 2x_2 & \le & 4 \\ x_1 \qquad & \ge & 0. \end{array}
\end{array}
$$

Если мы поменяем знаки коэффициентов целевой функции, то получим следующую задачу:

$$
\begin{array}{ll}
\text{максимизировать} & 2x_1 - 3x_2 \\
\text{при условиях} & \\
& \begin{array}{rcl} x_1 + \;\; x_2 & = & 7 \\ x_1 - 2x_2 & \le & 4 \\ x_1 \qquad & \ge & 0. \end{array}
\end{array}
$$

Теперь покажем, как преобразовать задачу линейного программирования, в которой на некоторые переменные не наложены ограничения неотрицательности, в задачу, в которой все переменные подчиняются этому условию. Предположим, что для некоторой переменной x_j ограничение неотрицательности отсутствует. Заменим все вхождения переменной x_j выражением $x'_j - x''_j$ и добавим ограничения неотрицательности $x'_j \ge 0$ и $x''_j \ge 0$. Так, если целевая функция содержит слагаемое $c_j x_j$, то оно заменяется на $c_j x'_j - c_j x''_j$, а если ограничение i содержит слагаемое $a_{ij} x_j$, оно заменяется на $a_{ij} x'_j - a_{ij} x''_j$. Любое допустимое решение $\widehat{x}$ новой задачи линейного программирования соответствует допустимому решению исходной задачи с $\bar{x}_j = \widehat{x}'_j - \widehat{x}''_j$ и тем же самым целевым значением. Точно так же и любое допустимое решение $\bar{x}$ исходной задачи линейного программирования соответствует допустимому решению $\widehat{x}$ новой задачи линейного программирования с $\widehat{x}'_j = \bar{x}_j$ и $\widehat{x}''_j = 0$, если $\bar{x}_j \ge 0$, или с $\widehat{x}''_j = -\bar{x}_j$ и $\widehat{x}'_j = 0$, если $\bar{x}_j < 0$. Две указанные задачи линейного программирования имеют одно и то же целевое значение независимо от знака $\bar{x}_j$. Следовательно, эти две задачи эквивалентны. Применив эту схему преобразования ко всем переменным, для которых нет ограничений неотрицательности, получим эквивалентную задачу линейного программирования, в которой на все переменные наложены ограничения неотрицательности.

Продолжая рассмотрение нашего примера, проверяем, для всех ли переменных есть соответствующие ограничения неотрицательности. Для переменной x_1

такое ограничение есть, а для переменной x_2 — нет. Заменив переменную x_2 переменными x'_2 и x''_2 и выполнив соответствующие преобразования, получим следующую задачу:

$$\begin{aligned} &\text{максимизировать} && 2x_1 - 3x'_2 + 3x''_2 \\ &\text{при условиях} && \\ & && x_1 + x'_2 - x''_2 = 7 \\ & && x_1 - 2x'_2 + 2x''_2 \le 4 \\ & && x_1, x'_2, x''_2 \ge 0. \end{aligned} \quad (29.22)$$

Теперь преобразуем ограничения-равенства в ограничения-неравенства. Предположим, что задача линейного программирования содержит ограничение-равенство $f(x_1, x_2, \ldots, x_n) = b$. Поскольку $x = y$ тогда и только тогда, когда справедливы оба неравенства, $x \ge y$ и $x \le y$, можно заменить данное ограничение-равенство парой ограничений-неравенств $f(x_1, x_2, \ldots, x_n) \le b$ и $f(x_1, x_2, \ldots, x_n) \ge b$. Выполнив такое преобразование для всех ограничений-равенств, получим задачу линейного программирования, в которой все ограничения являются неравенствами.

Наконец можно преобразовать ограничения вида "больше или равно" в ограничения вида "меньше или равно" путем умножения этих ограничений на -1. Любое ограничение вида

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i$$

эквивалентно ограничению

$$\sum_{j=1}^{n} -a_{ij} x_j \le -b_i \ .$$

Таким образом, заменив каждый коэффициент a_{ij} на $-a_{ij}$ и каждое значение b_j на $-b_j$, мы получим эквивалентное ограничение вида "меньше или равно".

Чтобы завершить преобразование нашего примера, заменим ограничение-равенство (29.22) двумя неравенствами и получим

$$\begin{aligned} &\text{максимизировать} && 2x_1 - 3x'_2 + 3x''_2 \\ &\text{при условиях} && \\ & && x_1 + x'_2 - x''_2 \le 7 \\ & && x_1 + x'_2 - x''_2 \ge 7 \\ & && x_1 - 2x'_2 + 2x''_2 \le 4 \\ & && x_1, x'_2, x''_2 \ge 0. \end{aligned} \quad (29.23)$$

Теперь изменим знак ограничения (29.23). Для единообразия имен переменных переименуем x_2' в x_2, а x_2'' — в x_3. Полученная стандартная форма имеет вид

$$\text{максимизировать} \quad 2x_1 - 3x_2 + 3x_3 \tag{29.24}$$

при условиях

$$x_1 + x_2 - x_3 \le 7 \tag{29.25}$$

$$-x_1 - x_2 + x_3 \le -7 \tag{29.26}$$

$$x_1 - 2x_2 + 2x_3 \le 4 \tag{29.27}$$

$$x_1, x_2, x_3 \ge 0. \tag{29.28}$$

Преобразование задач линейного программирования в каноническую форму

Чтобы эффективно решать задачу линейного программирования с помощью симплекс-метода, удобно записать ее в такой форме, когда некоторые ограничения заданы в виде равенств. Говоря более точно, мы будем приводить задачу к форме, в которой только ограничения неотрицательности заданы в виде неравенств, а остальные ограничения являются равенствами. Пусть ограничение-неравенство имеет вид

$$\sum_{j=1}^{n} a_{ij}x_j \le b_i \ . \tag{29.29}$$

Введем новую переменную s и перепишем неравенство (29.29) в виде двух ограничений:

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j \ , \tag{29.30}$$

$$s \ge 0 \ . \tag{29.31}$$

Переменная s называется ***вспомогательной переменной*** (slack variable), так как она определяет ***разность*** (slack) между левой и правой частями выражения (29.29). (Вскоре вы узнаете, почему удобно записывать ограничения в виде, когда в левой части находятся только вспомогательные переменные.) Поскольку неравенство (29.29) верно тогда и только тогда, когда одновременно выполнены равенство (29.30) и неравенство (29.31), можно применить данное преобразование ко всем ограничениям-неравенствам задачи линейного программирования и получить эквивалентную задачу, в которой в виде неравенств записаны только условия неотрицательности. При переходе от стандартной формы к канонической мы будем использовать для связанной с i-м ограничением вспомогательной переменной обозначение x_{n+i} (вместо s). Тогда i-е ограничение будет записано в виде равенства

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j \tag{29.32}$$

наряду с ограничением неотрицательности $x_{n+i} \ge 0$.

Применив данное преобразование ко всем ограничениям задачи линейного программирования в стандартной форме, получаем задачу в канонической форме. Например, для задачи, заданной формулами (29.24)–(29.28), введя вспомогательные переменные x_4, x_5 и x_6, получим

$$\text{максимизировать} \quad 2x_1 - 3x_2 + 3x_3 \tag{29.33}$$

при условиях

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.34}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.35}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.36}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \, . \tag{29.37}$$

В этой задаче линейного программирования все ограничения, за исключением условий неотрицательности, являются равенствами и все переменные подчиняются ограничениям неотрицательности. В записи каждого ограничения-равенства в левой части находится одна переменная, а остальные переменные находятся в правой части. Более того, в правой части каждого уравнения содержатся одни и те же переменные, и это только те переменные, которые входят в целевую функцию. Переменные, находящиеся в левой части равенств, называются ***базисными переменными*** (basic variables), а переменные, находящиеся в правой части, – ***небазисными переменными*** (nonbasic variables).

В задачах линейного программирования, удовлетворяющих указанным условиям, мы будем иногда опускать слова "максимизировать" и "при условиях", а также явные ограничения неотрицательности. Для обозначения значения целевой функции мы будем использовать переменную z. Полученная форма записи называется ***канонической формой*** (slack form). Если записать задачу линейного программирования (29.33)–(29.37) в канонической форме, можно получить

$$z = 2x_1 - 3x_2 + 3x_3 \tag{29.38}$$

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.39}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.40}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \, . \tag{29.41}$$

Для канонической формы, как и для стандартной, удобно иметь более короткий способ записи. Как будет показано в разделе 29.3, множества базисных и небазисных переменных в процессе работы симплекс-алгоритма будут меняться. Обозначим множество индексов небазисных переменных как N, а множество индексов базисных переменных – как B. Всегда выполняются соотношения $|N| = n$, $|B| = m$ и $N \cup B = \{1, 2, \ldots, n+m\}$. Уравнения будут индексироваться элементами множества B, а переменные правых частей будут индексироваться элементами множества N. Как и в стандартной форме, мы используем b_j, c_j и a_{ij} для обозначения констант и коэффициентов. Для обозначения необязательного постоянного члена в целевой функции используется v (позже вы узнаете, что включение константного члена в целевую функцию упрощает определение ее значения). Таким образом, можно кратко описать каноническую форму с помо-

щью кортежа (N, B, A, b, c, v); она служит кратким обозначением канонической формы

$$z = v + \sum_{j \in N} c_j x_j \tag{29.42}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{для } i \in B , \tag{29.43}$$

в которой все переменные x подчиняются условиям неотрицательности. Поскольку сумма $\sum_{j \in N} a_{ij} x_j$ в выражении (29.43) вычитается, значения элементов a_{ij} противоположны коэффициентам, входящим в каноническую форму.

Например, для канонической формы

$$\begin{aligned} z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 &= \ 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 &= \ 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2} \end{aligned}$$

мы имеем $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix} ,$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} ,$$

$c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^{\mathrm{T}}$ и $v = 28$. Заметим, что индексы у A, b и c необязательно являются множествами последовательных целых чисел; они зависят от того, какие индексы входят в множества B и N. В качестве примера противоположности элементов матрицы A коэффициентам канонической формы заметим, что в уравнение для x_1 входит слагаемое $x_3/6$, так что коэффициент a_{13} равен $-1/6$, а не $+1/6$.

Упражнения

29.1.1

Если записать задачу линейного программирования (29.24)–(29.28) в компактном виде (29.19)–(29.21), то чему будут равны n, m, A, b и c?

29.1.2

Укажите три допустимых решения задачи линейного программирования (29.24)–(29.28). Чему равно целевое значение для каждого решения?

29.1.3

Каковы N, B, A, b, c и v для канонической формы (29.38)–(29.41)?

29.1.4

Приведите следующую задачу линейного программирования к стандартной форме:

$$\begin{array}{lrcl}
\text{минимизировать} & 2x_1 + 7x_2 + x_3 & & \\
\text{при условиях} & & & \\
& x_1 \qquad\quad - x_3 & = & 7 \\
& 3x_1 + x_2 \qquad\quad & \geq & 24 \\
& x_2 & \geq & 0 \\
& x_3 & \leq & 0.
\end{array}$$

29.1.5

Приведите следующую задачу линейного программирования к канонической форме:

$$\begin{array}{lrcl}
\text{максимизировать} & 2x_1 \qquad\quad - 6x_3 & & \\
\text{при условиях} & & & \\
& x_1 + x_2 - x_3 & \leq & 7 \\
& 3x_1 - x_2 \qquad\quad & \geq & 8 \\
& -x_1 + 2x_2 + 2x_3 & \geq & 0 \\
& x_1, x_2, x_3 & \geq & 0.
\end{array}$$

Какие переменные являются базисными, а какие небазисными?

29.1.6

Покажите, что следующая задача линейного программирования является неразрешимой:

$$\begin{array}{lrcl}
\text{максимизировать} & 3x_1 - 2x_2 & & \\
\text{при условиях} & & & \\
& x_1 + x_2 & \leq & 2 \\
& -2x_1 - 2x_2 & \leq & -10 \\
& x_1, x_2 & \geq & 0.
\end{array}$$

29.1.7

Покажите, что следующая задача линейного программирования является неограниченной:

$$
\begin{array}{lrcl}
\text{максимизировать} & x_1 - x_2 & & \\
\text{при условиях} & & & \\
& -2x_1 + x_2 & \le & -1 \\
& -x_1 - 2x_2 & \le & -2 \\
& x_1, x_2 & \ge & 0.
\end{array}
$$

29.1.8

Пусть имеется задача линейного программирования общего вида с n переменными и m ограничениями. Предположим, что мы преобразовали ее в стандартную форму. Укажите верхнюю границу числа переменных и ограничений в полученной задаче.

29.1.9

Приведите пример задачи линейного программирования, для которой допустимая область является неограниченной, но оптимальное целевое значение конечно.

29.2. Формулировка задач в виде задач линейного программирования

Хотя основное внимание в данной главе уделяется симплекс-алгоритму, важно также понимать, в каких случаях задачу можно сформулировать в виде задачи линейного программирования. После того как задача сформулирована в этом виде, ее можно решить за полиномиальное время с помощью эллипсоидного алгоритма, или алгоритма внутренней точки. Существует несколько пакетов прикладных программ, эффективно решающих задачи линейного программирования, а значит, если проблему удастся сформулировать в виде задачи линейного программирования, то ее можно решить на практике с помощью такого пакета.

Рассмотрим несколько конкретных примеров задач линейного программирования. Начнем с двух уже рассмотренных ранее задач: задачи поиска кратчайших путей из одной вершины (см. главу 24) и задачи поиска максимального потока (см. главу 26). После этого мы опишем задачу поиска потока с минимальной стоимостью. Для данной задачи существует алгоритм с полиномиальными затратами времени, не основанный на линейном программировании, однако мы не будем его рассматривать. И наконец мы опишем задачу многопродуктного потока (multicomodity-flow problem), единственный известный полиномиальный по времени алгоритм решения которой базируется на линейном программировании.

При решении задач с графами в части VI мы использовали запись атрибутов в виде $v.d$ и $(u, v).f$. Однако линейное программирование обычно использует не присоединенные атрибуты, а переменные с нижними индексами. Таким образом, выражая переменные в задачах линейного программирования, необходимо указы-

вать вершины и ребра с помощью индексов. Например, вес кратчайшего пути для вершины v обозначается не как $v.d$, а как d_v. Аналогично поток из вершины u в вершину v обозначается не как $(u,v).f$, а как f_{uv}. Для величин, являющихся входными данными для задач, таких как веса или пропускные способности ребер, мы будем продолжать использовать записи $w(u,v)$ и $c(u,v)$.

Кратчайшие пути

Задачу поиска кратчайших путей из одной вершины-источника можно сформулировать в виде задачи линейного программирования. В данном разделе мы остановимся на формулировке задачи кратчайшего пути для одной пары, а более общий случай задачи поиска кратчайших путей из одного источника предлагается рассмотреть в качестве упр. 29.2.3.

В задаче поиска кратчайшего пути для одной пары у нас имеются взвешенный ориентированный граф $G = (V, E)$, весовая функция $w : E \to \mathbb{R}$, ставящая в соответствие ребрам графа действительные веса, исходная вершина s и вершина назначения t. Мы хотим вычислить значение d_t, которое является весом кратчайшего пути из s в t. Чтобы записать эту задачу в виде задачи линейного программирования, необходимо определить множество переменных и ограничения, которые позволят определить, какой путь из s в t является кратчайшим. К счастью, алгоритм Беллмана–Форда именно это и делает. Когда алгоритм Беллмана–Форда завершает свою работу, для каждой вершины v оказывается вычисленным значение d_v (мы используем запись с индексом, а не атрибут), такое, что для каждого ребра $(u,v) \in E$ выполняется условие $d_v \le d_u + w(u,v)$. Исходная вершина изначально получает значение $d_s = 0$, которое никогда не изменяется. Таким образом, мы получаем следующую задачу линейного программирования для вычисления веса кратчайшего пути из s в t:

$$\text{максимизировать} \quad d_t \tag{29.44}$$

при условиях

$$d_v \le d_u + w(u,v) \quad \text{для каждого ребра } (u,v) \in E \,, \tag{29.45}$$

$$d_s = 0 \,. \tag{29.46}$$

Вас может удивить то, что данная задача линейного программирования максимизирует целевую функцию, в то время как предполагается, что задача вычисляет кратчайшие пути. Мы не хотим минимизировать целевую функцию, поскольку тогда присваивание $\bar{d}_v = 0$ для всех $v \in V$ даст оптимальное решение задачи линейного программирования без решения задачи поиска кратчайшего пути. Мы выполняем максимизацию, поскольку оптимальное решение задачи поиска кратчайших путей присваивает каждому $\bar{d}_v$ значение $\min_{u:(u,v)\in E} \{\bar{d}_u + w(u,v)\}$, так что $\bar{d}_v$ является наибольшим значением, которое не превышает все значения множества $\{\bar{d}_u + w(u,v)\}$. Мы хотим максимизировать d_v для всех вершин v на кратчайшем пути из s в t при условии выполнения указанных ограничений для всех вершин v, и максимизация d_t достигает этой цели.

Эта задача линейного программирования имеет $|V|$ переменных d_v, по одной для каждой вершины $v \in V$. В ней также имеется $|E|+1$ ограничений: по одному

для каждого ребра плюс дополнительное ограничение, что вес исходной вершины кратчайшего пути всегда имеет значение 0.

Максимальный поток

Задачу поиска максимального потока также можно сформулировать в виде задачи линейного программирования. Напомним, что в ней задаются ориентированный граф $G = (V, E)$, в котором каждое ребро $(u, v) \in E$ имеет неотрицательную пропускную способность $c(u, v) \geq 0$, и две различные вершины, источник s и сток t. Согласно определению из раздела 26.1 поток является действительной функцией $f : V \times V \to \mathbb{R}$, удовлетворяющей ограничениям пропускной способности и сохранения потока. Максимальный поток представляет собой поток, который удовлетворяет данным ограничениям и максимизирует величину потока, представляющую собой суммарный поток, выходящий из источника, минус суммарный поток, входящий в источник. Таким образом, поток удовлетворяет линейным ограничениям, а величина потока является линейной функцией. Напомним также, что мы предполагаем, что $c(u, v) = 0$, если $(u, v) \notin E$. Теперь можно записать задачу максимизации потока в виде задачи линейного программирования:

$$\text{максимизировать} \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{29.47}$$

при условиях

$$f_{uv} \leq c(u, v) \quad \text{для всех } u, v \in V\ , \tag{29.48}$$

$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \quad \text{для каждой вершины } u \in V - \{s, t\}\ , \tag{29.49}$$

$$f_{uv} \geq 0 \quad \text{для всех } u, v \in V\ . \tag{29.50}$$

Эта задача линейного программирования имеет $|V|^2$ переменных, соответствующих потоку между каждой парой вершин, и $2|V|^2 + |V| - 2$ ограничений.

Обычно более эффективно решаются задачи линейного программирования меньшей размерности. В задаче (29.47)–(29.50) для простоты записи считается, что поток и пропускная способность каждой пары вершин u, v, таких, что $(u, v) \notin E$, равны 0. Но было бы эффективнее переписать задачу так, чтобы в ней содержалось $O(V + E)$ ограничений, что и предлагается сделать в упр. 29.2.5.

Поиск потока минимальной стоимости

До сих пор в данном разделе использовалось линейное программирование для решения задач, для которых уже известны эффективные алгоритмы. Фактически хороший алгоритм, разработанный специально для конкретной задачи, например алгоритм Дейкстры для задачи поиска кратчайшего пути из одной вершины или метод проталкивания для задачи максимального потока, обычно более эффективен, чем линейное программирование, — как в теории, так и на практике.

Истинная ценность линейного программирования состоит в возможности решения новых задач. Вспомним задачу подготовки к предвыборной кампании, описанную в начале данной главы. Задача получения необходимого количества

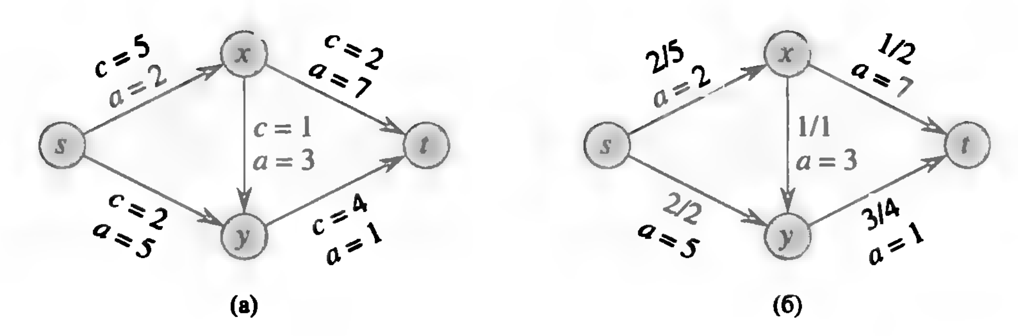


Рис. 29.3. **(а)** Пример задачи поиска потока минимальной стоимости. Пропускные способности обозначены как c, а стоимости — как a. Вершина s является источником, а вершина t — стоком, и необходимо переслать четыре единицы потока из s в t. **(б)** Решение задачи поиска потока минимальной стоимости, в которой из истока s в сток t пересылаются четыре единицы потока. Для каждого ребра поток и пропускная способность указаны в виде "поток/пропускная способность".

голосов при минимальных затратах не решается ни одним из алгоритмов, уже изученных нами в данной книге, однако ее можно решить с помощью линейного программирования. О реальных задачах, которые можно решить с помощью линейного программирования, написано множество книг. Линейное программирование также чрезвычайно полезно при решении разновидностей задач, для которых еще не известны эффективные алгоритмы.

Рассмотрим, например, следующее обобщение задачи поиска максимального потока. Предположим, что каждому ребру (u, v), помимо пропускной способности $c(u, v)$, соответствует некая стоимость $a(u, v)$, принимающая действительные значения. Как и в задаче поиска максимального потока, мы считаем, что $c(u, v) = 0$, если $(u, v) \notin E$, и что антипараллельных ребер в графе нет. Если по ребру (u, v) послано f_{uv} единиц потока, это приводит к стоимости пересылки $a(u, v)f_{uv}$. Также задано целевое значение потока d. Необходимо переправить d единиц потока из s в t таким образом, чтобы общая стоимость, связанная с передачей потока, $\sum_{(u,v)\in E} a(u, v)f_{uv}$, была минимальной. Эта задача называется ***задачей поиска потока минимальной стоимости***.

На рис. 29.3, (а) представлен пример задачи поиска потока минимальной стоимости. Нужно переслать четыре единицы потока из s в t, минимизировав суммарную стоимость (пропускная способность каждого ребра обозначена как c, а стоимость — как a). С любым допустимым потоком, т.е. с функцией f, удовлетворяющей ограничениям (29.48)–(29.50), связана стоимость $\sum_{(u,v)\in E} a(u, v)f_{uv}$. Необходимо найти такой поток, который минимизирует эту стоимость. Оптимальное решение показано на рис. 29.3, (б); ему соответствует суммарная стоимость $\sum_{(u,v)\in E} a(u, v)f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

Существуют алгоритмы с полиномиальными затратами времени, специально разработанные для задачи поиска потока минимальной стоимости, но они выходят за рамки данной книги. Запишем рассматриваемую задачу в виде задачи линейного программирования. Эта задача линейного программирования напоминает задачу, записанную для максимизации потока, однако содержит дополнительное ограничение, состоящее в том, что величина потока составляет ровно d единиц,

и новую целевую функцию минимизации стоимости:

$$\text{минимизировать} \quad \sum_{(u,v)\in E} a(u,v) f_{uv} \tag{29.51}$$

при условиях

$$f_{uv} \le c(u,v) \quad \text{для всех } u,v \in V \;,$$

$$\sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} = 0 \quad \text{для каждой вершины } u \in V - \{s,t\} \;,$$

$$\sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} = d \;,$$

$$f_{uv} \ge 0 \quad \text{для всех } u,v \in V \;. \tag{29.52}$$

Многопродуктный поток

В качестве последнего примера рассмотрим еще одну задачу поиска потоков. Предположим, что компания Lucky Puck из раздела 26.1 приняла решение о расширении ассортимента продукции и отгружает не только хоккейные шайбы, но также клюшки и шлемы. Каждый элемент снаряжения выпускается на отдельной фабрике, хранится на отдельном складе и должен ежедневно доставляться с фабрики на склад. Клюшки производятся в Ванкувере и доставляются в Саскатун, а шлемы производятся в Эдмонтоне и доставляются в Реджину. Однако пропускная способность сети доставки не изменилась, и различные элементы, или ***продукты***, должны совместно использовать одну и ту же сеть.

Данный пример представляет собой экземпляр задачи ***многопродуктного потока*** (multicommodity flow). В этой задаче снова задан ориентированный граф $G = (V, E)$, в котором каждое ребро $(u, v) \in E$ имеет неотрицательную пропускную способность $c(u, v) \ge 0$. Как и в задаче поиска максимального потока, неявно подразумевается, что $c(u, v) = 0$ для $(u, v) \notin E$ и что в графе нет антипараллельных ребер. Кроме того, даны k различных продуктов $K_1, K_2, \ldots, K_k$, причем каждый i-й продукт характеризуется тройкой $K_i = (s_i, t_i, d_i)$, где s_i — источник продукта i, t_i — место назначения продукта i, а d_i — спрос, т.е. желаемая величина потока продукта i из s_i в t_i. По определению поток продукта i, обозначаемый f_i (так что f_{iuv} представляет собой поток продукта i из вершины u в вершину v), — это действительная функция, удовлетворяющая ограничениям сохранения потока и пропускной способности. Определим ***совокупный поток*** (aggregate flow) f_{uv} как сумму потоков различных продуктов: $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. Совокупный поток по ребру (u, v) не должен превышать пропускную способность ребра (u, v). При таком способе описания задачи не нужно ничего минимизировать; необходимо только определить, можно ли найти такой поток. Поэтому мы записываем

задачу линейного программирования с “пустой” целевой функцией:

$$
\begin{array}{lll}
\text{минимизировать} & 0 & \\
\text{при условиях} & & \\
& \displaystyle\sum_{i=1}^{k} f_{iuv} \leq c(u,v) & \text{для всех } u,v \in V\ , \\
& \displaystyle\sum_{v\in V} f_{iuv} - \sum_{v\in V} f_{ivu} = 0 & \text{для всех } i = 1,2,\ldots,k \text{ и} \\
& & \text{для всех } u \in V - \{s_i, t_i\}\ , \\
& \displaystyle\sum_{v\in V} f_{i,s_i,v} - \sum_{v\in V} f_{i,v,s_i} = d_i & \text{для всех } i = 1,2,\ldots,k\ , \\
& f_{iuv} \geq 0 & \text{для всех } u,v \in V \text{ и} \\
& & \text{для всех } i = 1,2,\ldots,k\ .
\end{array}
$$

Единственный известный алгоритм решения этой задачи с полиномиальным временем выполнения состоит в том, чтобы записать ее в виде задачи линейного программирования, а затем решить с помощью полиномиального по времени алгоритма линейного программирования.

Упражнения

29.2.1

Приведите задачу линейного программирования поиска кратчайшего пути для одной пары вершин, заданную формулами (29.44)–(29.46), к стандартной форме.

29.2.2

Запишите в явном виде задачу линейного программирования, соответствующую задаче поиска кратчайшего пути из узла s в узел y на рис. 24.2, (а).

29.2.3

В задаче поиска кратчайшего пути из одной вершины необходимо найти веса кратчайших путей из вершины s во все вершины $v \in V$. Запишите задачу линейного программирования для данного графа G, решение которой обладает тем свойством, что d_v является весом кратчайшего пути из s в v для всех вершин $v \in V$.

29.2.4

Запишите задачу линейного программирования, соответствующую поиску максимального потока на рис. 26.1, (а).

29.2.5

Перепишите задачу линейного программирования для поиска максимального потока (29.47)–(29.50) так, чтобы в ней было только $O(V + E)$ ограничений.

29.2.6

Сформулируйте задачу линейного программирования, которая для заданного двудольного графа $G = (V, E)$ решает задачу о взвешенных паросочетаниях с максимальным весом (maximum-bipartite-matching).

29.2.7

В задаче поиска ***многопродуктного потока с минимальной стоимостью*** (minimum-cost multicommodity-flow problem) задан ориентированный граф $G = (V, E)$, в котором каждому ребру $(u, v) \in E$ соответствуют неотрицательная пропускная способность $c(u, v) \geq 0$ и стоимость $a(u, v)$. Как и в задаче многопродуктного потока, имеется k различных товаров $K_1, K_2, \ldots, K_k$, при этом каждый продукт i характеризуется тройкой $K_i = (s_i, t_i, d_i)$. Поток f_i для i-го продукта и совокупный поток f_{uv} вдоль ребра (u, v) определяются так же, как и в задаче многопродуктного потока. Допустимым является такой поток, для которого совокупный поток вдоль каждого ребра (u, v) не превышает его пропускной способности. Стоимость потока определяется как $\sum_{u,v \in V} a(u, v) f_{uv}$, и цель состоит в том, чтобы найти допустимый поток с минимальной стоимостью. Запишите данную задачу в виде задачи линейного программирования.

29.3. Симплекс-алгоритм

Симплекс-алгоритм является классическим методом решения задач линейного программирования. В отличие от многих других приведенных в данной книге алгоритмов, время выполнения этого алгоритма в худшем случае не является полиномиальным. Однако он действительно выражает суть линейного программирования и на практике зачастую бывает замечательно быстрым.

В дополнение к описанной ранее геометрической интерпретации можно провести определенные аналогии между ним и методом исключения Гаусса, рассмотренным в разделе 28.1. Метод исключения Гаусса применяется при поиске решения системы линейных уравнений. На каждой итерации система переписывается в эквивалентной форме, имеющей определенную структуру. После ряда итераций получается система, записанная в таком виде, что можно легко найти ее решение. Симплекс-алгоритм работает аналогичным образом, и его можно рассматривать как метод исключения Гаусса для неравенств.

Опишем основную идею, лежащую в основе итераций симплекс-алгоритма. С каждой итерацией связывается некое "базисное решение", которое легко получить из канонической формы задачи линейного программирования: каждой небазисной переменной присваивается значение 0, и из ограничений-равенств вычисляются значения базисных переменных. Базисное решение всегда соответствует некой вершине симплекса. С алгебраической точки зрения каждая итерация преобразует одну каноническую форму в эквивалентную. Целевое значение, соответствующее новому базисному допустимому решению, должно быть не меньше (как правило, больше) целевого значения предыдущей итерации. Чтобы добиться

этого увеличения, выбирается некоторая небазисная переменная, причем такая, что при увеличении ее значения от нуля целевое значение также увеличится. То, насколько можно увеличить данную переменную, определяется другими ограничениями, а именно — ее значение увеличивается до тех пор, пока какая-либо базисная переменная не станет равной нулю. После этого каноническая форма переписывается так, что эта базисная переменная и выбранная небазисная переменная меняются ролями. Хотя мы использовали определенные начальные значения переменных для описания алгоритма и будем использовать их в наших доказательствах, в алгоритме данное решение в явном виде не поддерживается. Он просто переписывает задачу линейного программирования до тех пор, пока оптимальное решение не станет "очевидным".

Пример симплекс-алгоритма

Начнем с конкретного примера. Рассмотрим следующую задачу линейного программирования в стандартной форме:

максимизировать
$$3x_1 + x_2 + 2x_3 \tag{29.53}$$
при условиях
$$x_1 + x_2 + 3x_3 \le 30 \tag{29.54}$$
$$2x_1 + 2x_2 + 5x_3 \le 24 \tag{29.55}$$
$$4x_1 + x_2 + 2x_3 \le 36 \tag{29.56}$$
$$x_1, x_2, x_3 \ge 0. \tag{29.57}$$

Чтобы можно было применить симплекс-алгоритм, необходимо преобразовать данную задачу в каноническую форму, как описано в разделе 29.1. Вспомогательные переменные — это не просто элементы алгебраического преобразования, они являются содержательным алгоритмическим понятием. Напомним (см. раздел 29.1), что каждой переменной соответствует ограничение неотрицательности. Будем говорить, что ограничение-равенство является ***строгим*** (tight) при определенном наборе значений его небазисных переменных, если при этих значениях базисная переменная данного ограничения становится равной нулю. Аналогично набор значений небазисных переменных, который делает базисную переменную отрицательной, ***нарушает*** данное ограничение. Таким образом, вспомогательные переменные наглядно показывают, насколько сильно каждое ограничение отличается от строгого, и тем самым помогают определить, насколько можно увеличить значения небазисных переменных, не нарушив ни одного ограничения.

Свяжем с ограничениями (29.54)–(29.56) вспомогательные переменные x_4, x_5 и x_6 соответственно и приведем задачу линейного программирования к канонической форме. В результате получится следующая задача:

$$z = 3x_1 + x_2 + 2x_3 \tag{29.58}$$
$$x_4 = 30 - x_1 - x_2 - 3x_3 \tag{29.59}$$
$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \tag{29.60}$$
$$x_6 = 36 - 4x_1 - x_2 - 2x_3 . \tag{29.61}$$

Система ограничений (29.59)–(29.61) содержит три уравнения и шесть переменных. Любое задание значений переменных x_1, x_2 и x_3 определяет значения переменных x_4, x_5 и x_6, следовательно, существует бесконечное число решений данной системы уравнений. Решение является допустимым, если все $x_1, x_2, \ldots, x_6$ неотрицательны. Число допустимых решений также может быть бесконечным. Свойство бесконечности числа возможных решений подобной системы понадобится нам в дальнейших доказательствах. Рассмотрим ***базисное решение*** (basic solution): установим все (небазисные) переменные правой части равными нулю и вычислим значения (базисных) переменных левой части. В данном примере базисным решением является $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$, и ему соответствует целевое значение $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Заметим, что в этом базисном решении $\bar{x}_i = b_i$ для всех $i \in B$. Итерация симплекс-алгоритма переписывает множество уравнений и целевую функцию так, что в правой части оказывается другое множество переменных. Таким образом, с переписанной задачей связано другое базисное решение. Мы подчеркиваем, что такая перезапись никоим образом не меняет лежащую в основе задачу линейного программирования; задача на каждой итерации имеет точно то же множество допустимых решений, что и задача на предыдущей итерации. Однако эта задача имеет базисное решение, отличное от базисного решения предыдущей итерации.

Если базисное решение является также допустимым, оно называется ***допустимым базисным решением***. В процессе работы симплекс-алгоритма базисное решение практически всегда будет допустимым базисным решением. Однако в разделе 29.5 мы покажем, что в нескольких первых итерациях симплекс-алгоритма базисное решение может не быть допустимым.

На каждой итерации нашей целью является переформулировка задачи линейного программирования таким образом, чтобы новое базисное решение имело большее целевое значение. Мы выбираем некоторую небазисную переменную x_e, коэффициент при которой в целевой функции положителен, и увеличиваем ее значение настолько, насколько это возможно без нарушения существующих ограничений. Переменная x_e становится базисной, а некоторая другая переменная x_l становится небазисной. Значения остальных базисных переменных и целевой функции также могут измениться.

Продолжая изучение примера, рассмотрим возможность увеличения значения x_1. При увеличении x_1 значения переменных x_4, x_5 и x_6 уменьшаются. Поскольку на каждую переменную наложено ограничение неотрицательности, ни одна из этих переменных не должна стать отрицательной. Если x_1 увеличить более чем на 30, то x_4 станет отрицательным, а x_5 и x_6 станут отрицательными при увеличении x_1 на 12 и 9 соответственно. Третье ограничение (29.61) является самым строгим, именно оно определяет, насколько можно увеличить x_1. Следовательно, меняем ролями переменные x_1 и x_6. Решив уравнение (29.61) относительно x_1, получим

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} . \qquad (29.62)$$

Чтобы записать другие уравнения с x_6 в правой части, подставим вместо x_1 выражение из (29.62). Для уравнения (29.59) получаем

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \,. \end{aligned} \tag{29.63}$$

Аналогично комбинируем уравнение (29.62) с ограничением (29.60) и целевой функцией (29.58) и записываем нашу задачу линейного программирования в следующем виде:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{29.64}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.65}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.66}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \,. \tag{29.67}$$

Эта операция называется ***замещением***. Как было показано выше, в процессе замещения выбираются небазисная переменная x_e, называемая ***вводимой переменной*** (entering variable), и базисная переменная x_l, называемая ***выводимой переменной*** (leaving variable), которые затем меняются ролями.

Задача, записанная уравнениями (29.64)–(29.67), эквивалентна задаче 29.58–29.61. В процессе работы симплекс-алгоритма выполняются только операции переноса переменных из левой части уравнения в правую и обратно, а также подстановки одного уравнения в другое. Первая операция очевидным образом создает эквивалентную задачу, то же самое можно сказать и о второй операции (см. упр. 29.3.3).

Чтобы продемонстрировать эквивалентность указанных задач, убедимся, что исходное базисное решение $(0, 0, 0, 30, 24, 36)$ удовлетворяет новым уравнениям (29.65)–(29.67) и имеет целевое значение $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$. В базисном решении, связанном с новой задачей, новые небазисные переменные равны нулю. Таким образом, оно имеет вид $(9, 0, 0, 21, 6, 0)$, а соответствующее целевое значение $z = 27$. Простые арифметические действия позволяют убедиться, что данное решение также удовлетворяет уравнениям (29.59)–(29.61) и при подстановке в целевую функцию (29.58) имеет целевое значение $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$.

Продолжая рассмотрение примера, необходимо найти новую базисную переменную, значение которой можно увеличить. Нет смысла увеличивать x_6, поскольку при ее увеличении целевое значение уменьшается. Можно попробовать увеличить x_2 или x_3; мы выберем x_3. Насколько можно увеличить x_3, чтобы не нарушить ни одно из ограничений? Ограничение (29.65) допускает увеличение, не превышающее 18, ограничение (29.66) — $42/5$, а ограничение (29.67) — $3/2$.

Третье ограничение снова оказывается самым строгим, следовательно, мы переписываем его так, чтобы x_3 было в левой части, а x_5 — в правой. Затем подставляем это новое уравнение $x_3 = 3/2 - 3x_2/8 - x_5/4 + x_6/8$ в уравнения (29.64)–(29.66) и получаем новую эквивалентную задачу:

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.68}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.69}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.70}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \,. \tag{29.71}$$

С этой системой связано базисное решение $(33/4, 0, 3/2, 69/4, 0, 0)$ с целевым значением $111/4$. Теперь единственная возможность увеличить целевое значение — увеличить x_2. Имеющиеся ограничения задают верхние границы увеличения 132, 4 и ∞ соответственно. (Верхняя граница в ограничении (29.71) равна ∞, поскольку при увеличении x_2 значение базисной переменной x_4 также увеличивается. Следовательно, данное уравнение не налагает никаких ограничений на величину возможного увеличения x_2.) Увеличиваем x_2 до 4 и делаем эту переменную базисной. Затем решаем уравнение (29.70) относительно x_2, подставляем полученное выражения в другие уравнения и получаем новую задачу:

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.72}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.73}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.74}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \,. \tag{29.75}$$

В полученной задаче все коэффициенты целевой функции отрицательны. Как будет показано далее, такая ситуация возникает только тогда, когда базисное решение переписанной задачи линейного программирования является оптимальным ее решением. Таким образом, для данной задачи решение $(8, 4, 0, 18, 0, 0)$ с целевым значением 28 является оптимальным. Теперь можно вернуться к исходной задаче линейного программирования, заданной уравнениями (29.53)–(29.57). Исходная задача содержит только переменные x_1, x_2 и x_3, поэтому оптимальное решение имеет вид $x_1 = 8$, $x_2 = 4$ и $x_3 = 0$ с целевым значением $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Заметим, что значения вспомогательных переменных в окончательном решении показывают, насколько велик резерв в каждом неравенстве. Вспомогательная переменная x_4 равна 18, а значение левой части в неравенстве (29.54) равно $8 + 4 + 0 = 12$, что на 18 меньше, чем значение правой части этого неравенства — 30. Вспомогательные переменные x_5 и x_6 равны нулю, и действительно, в неравенствах (29.55) и (29.56) левые и правые части равны. Обратите внимание на то,

что, даже если коэффициенты исходной канонической формы являются целочисленными, коэффициенты последующих эквивалентных задач не обязательно целочисленны, как не обязательно целочисленны и промежуточные решения. Более того, окончательное решение задачи линейного программирования также не обязательно будет целочисленным; в данном примере это не более чем совпадение.

Замещение

Формализуем процедуру замещения. Процедура PIVOT получает в качестве входных данных каноническую форму задачи линейного программирования, заданную кортежем (N, B, A, b, c, v), индекс l выводимой переменной x_l и индекс e вводимой переменной x_e. Она возвращает кортеж $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$, описывающий новую каноническую форму. (Еще раз напомним, что элементы матриц A и $\widehat{A}$, имеющих размер $m \times n$, являются числами, обратными коэффициентам канонической формы.)

PIVOT(N, B, A, b, c, v, l, e)

1 // Вычисление коэффициентов уравнения для новой
  // базисной переменной x_e.
2 Пусть $\widehat{A}$ — новая матрица размером $m \times n$
3 $\widehat{b}_e = b_l / a_{le}$
4 **for** каждого $j \in N - \{e\}$
5     $\widehat{a}_{ej} = a_{lj} / a_{le}$
6 $\widehat{a}_{el} = 1/a_{le}$
7 // Вычисление коэффициентов остальных ограничений.
8 **for** каждого $i \in B - \{l\}$
9     $\widehat{b}_i = b_i - a_{ie}\widehat{b}_e$
10   **for** каждого $j \in N - \{e\}$
11         $\widehat{a}_{ij} = a_{ij} - a_{ie}\widehat{a}_{ej}$
12     $\widehat{a}_{il} = -a_{ie}\widehat{a}_{el}$
13 // Вычисление целевой функции.
14 $\widehat{v} = v + c_e\widehat{b}_e$
15 **for** каждого $j \in N - \{e\}$
16     $\widehat{c}_j = c_j - c_e\widehat{a}_{ej}$
17 $\widehat{c}_l = -c_e\widehat{a}_{el}$
18 // Вычисление новых множеств базисных
  // и небазисных переменных.
19 $\widehat{N} = N - \{e\} \cup \{l\}$
20 $\widehat{B} = B - \{l\} \cup \{e\}$
21 **return** $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$

Процедура PIVOT работает следующим образом. В строках 3–6 вычисляются коэффициенты нового уравнения для x_e; для этого уравнение, в левой части которого стоит x_l, переписывается так, чтобы в левой части оказалась x_e. В строках 7–11 оставшиеся уравнения обновляются путем подстановки правой части

полученного нового уравнения вместо всех вхождений x_e. В строках 8–12 такая же подстановка выполняется для целевой функции, а в строках 14–17 обновляются множества небазисных и базисных переменных. Строка 21 возвращает новую каноническую форму. Если $a_{le} = 0$, вызов процедуры PIVOT приведет к ошибке (деление на нуль), однако, как будет показано в ходе доказательства лемм 29.2 и 29.12, данная процедура вызывается только тогда, когда $a_{le} \neq 0$.

Рассмотрим, как процедура PIVOT действует на значения переменных базисного решения.

Лемма 29.1

Рассмотрим вызов PIVOT(N, B, A, b, c, v, l, e), в котором $a_{le} \neq 0$. Пусть значения, возвращаемые вызовом, представляют собой $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$ и пусть $\bar{x}$ обозначает базисное решение после выполнения вызова. Тогда

1. $\bar{x}_j = 0$ для каждого $j \in \widehat{N}$;
2. $\bar{x}_e = b_l/a_{le}$;
3. $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ для каждого $i \in \widehat{B} - \{e\}$.

Доказательство. Первое утверждение верно, поскольку в базисном решении все небазисные переменные задаются равными нулю. Если мы приравняем нулю все небазисные переменные в ограничении

$$x_i = \widehat{b}_i - \sum_{j \in \widehat{N}} \widehat{a}_{ij} x_j ,$$

то получим, что $\bar{x}_i = \widehat{b}_i$ для каждого $i \in \widehat{B}$. Поскольку $e \in \widehat{B}$, строка 3 процедуры PIVOT дает

$$\bar{x}_e = \widehat{b}_e = b_l/a_{le} ,$$

что доказывает второе утверждение. Аналогично, используя строку 9 для каждого $i \in \widehat{B} - \{e\}$, мы имеем

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e ,$$

что доказывает третье утверждение. ■

Формальный симплекс-алгоритм

Теперь мы готовы формализовать симплекс-алгоритм, который уже продемонстрировали на примере. Этот пример был очень удачным, однако еще необходимо ответить на следующие вопросы.

- Как определить, что задача линейного программирования является разрешимой?
- Что делать, если задача линейного программирования является разрешимой, однако начальное базисное решение не является допустимым?

- Как определить, что задача линейного программирования является неограниченной?
- Как выбирать вводимую и выводимую переменные?

В разделе 29.5 мы покажем, как определить, является ли задача разрешимой и, в случае положительного ответа, как найти каноническую форму, в которой начальное базисное решение является допустимым. На данном этапе предположим, что имеется процедура INITIALIZE-SIMPLEX(A, b, c), которая получает на входе задачу линейного программирования в стандартной форме, т.е. матрицу $A = (a_{ij})$ размером $m \times n$, m-мерный вектор $b = (b_i)$ и n-мерный вектор $c = (c_j)$. Если задача неразрешима, процедура возвращает соответствующее сообщение и завершается. В противном случае она возвращает каноническую форму, начальное базисное решение которой является допустимым.

Процедура SIMPLEX получает на входе задачу линейного программирования в стандартной форме, описанной выше. Она возвращает n-мерный вектор $\bar{x} = (\bar{x}_j)$, который является оптимальным решением задачи линейного программирования, заданной формулами (29.19)–(29.21).

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
 2  Пусть Δ — новый вектор длиной m
 3  while c_j > 0 для некоторого индекса j ∈ N
 4      Выбрать индекс e ∈ N, для которого c_e > 0
 5      for каждого индекса i ∈ B
 6          if a_ie > 0
 7              Δ_i = b_i/a_ie
 8          else Δ_i = ∞
 9      Выбрать индекс l ∈ B, который минимизирует Δ_l
10      if Δ_l == ∞
11          return "задача неограниченная"
12      else (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, e)
13  for i = 1 to n
14      if i ∈ B
15          x̄_i = b_i
16      else x̄_i = 0
17  return (x̄_1, x̄_2, ..., x̄_n)
```

Процедура SIMPLEX работает следующим образом. В строке 1 выполняется вызов упомянутой выше процедуры INITIALIZE-SIMPLEX(A, b, c), которая или определяет, что предложенная задача неразрешима, или возвращает каноническую форму, базисное решение которой является допустимым. Главная часть алгоритма содержится в цикле **while** в строках 3–12. Если все коэффициенты целевой функции отрицательны, цикл **while** завершается. В противном случае в строке 4 мы выбираем в качестве вводимой переменной некоторую переменную x_e, коэффициент при которой в целевой функции положителен. Хотя в качестве вводимой можно выбирать любую такую переменную, предполагается, что

используется некое предварительно заданное детерминистическое правило. Затем, в строках 5–9, выполняется проверка каждого ограничения и выбирается то, которое более всего лимитирует величину увеличения x_e, не приводящего к нарушению ограничений неотрицательности; базисная переменная, связанная с этим ограничением, выбирается в качестве выводимой переменной x_l. Если таких переменных несколько, можно выбрать любую из них, однако предполагается, что и здесь мы используем некое предварительно заданное детерминистическое правило. Если ни одно из ограничений не лимитирует возможность увеличения вводимой переменной, алгоритм выдает сообщение "задача неограниченная" (строка 11). В противном случае в строке 12 роли вводимой и выводимой переменных меняются путем вызова описанной выше процедуры PIVOT(N, B, A, b, c, v, l, e). В строках 13–16 вычисляется решение $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ исходной задачи линейного программирования путем присваивания всем небазисным переменным нулевого значения, всем базисным переменным $\bar{x}_i$ — соответствующих значений b_i, а строка 17 возвращает эти значения.

Чтобы показать, что процедура SIMPLEX работает корректно, сначала покажем, что если в процедуре задано начальное допустимое значение и она завершилась, то при этом или возвращается допустимое решение, или сообщается, что данная задача является неограниченной. Затем мы покажем, что процедура SIMPLEX завершается; и наконец, в разделе 29.4 (теорема 29.10), будет показано, что возвращаемое процедурой решение является оптимальным.

Лемма 29.2

Пусть задана задача линейного программирования (A, b, c); предположим, что вызываемая в строке 1 процедуры SIMPLEX процедура INITIALIZE-SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым. Тогда если процедура SIMPLEX возвращает решение в строке 17, это решение является допустимым решением задачи линейного программирования. Если процедура SIMPLEX выводит сообщение о неограниченности в строке 11, данная задача линейного программирования является неограниченной.

Доказательство. Воспользуемся следующим тройным инвариантом цикла:

в начале каждой итерации цикла **while** в строках 3–12

1. имеющаяся каноническая форма эквивалентна канонической форме, полученной в результате вызова процедуры INITIALIZE-SIMPLEX;
2. для всех $i \in B$ выполняется $b_i \geq 0$;
3. базисное решение, связанное с данной канонической формой, является допустимым.

Инициализация. Эквивалентность канонических форм для первой итерации очевидна. В формулировке леммы предполагается, что вызов процедуры INITIALIZE-SIMPLEX в строке 1 процедуры SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым. Следовательно, третье утверждение справедливо. Далее, поскольку в базисном решении каж-

дой базисной переменной x_i присваивается значение b_i, а допустимость базисного решения предполагает неотрицательность всех базисных переменных x_i, то $b_i \geq 0$. Таким образом, второе утверждение инварианта также справедливо.

Сохранение. Покажем, что данный инвариант цикла сохраняется при условии, что оператор **return** в строке 11 не выполняется. Случай выполнения этой строки мы рассмотрим при обсуждении завершения цикла.

Каждая итерация цикла **while** меняет ролями некоторую базисную и небазисную переменные с помощью вызова процедуры PIVOT. Согласно упр. 29.3.3 новая каноническая форма эквивалентна канонической форме из предыдущей итерации, которая, согласно инварианту цикла, эквивалентна исходной канонической форме.

Теперь покажем, что сохраняется вторая часть инварианта цикла. Предположим, что в начале каждой итерации цикла **while** для всех $i \in B$ выполняется соотношение $b_i \geq 0$, и покажем, что эти неравенства остаются верными после вызова процедуры PIVOT в строке 12. Поскольку изменения в переменные b_i и множество B вносятся только в этой строке, достаточно показать, что она сохраняет данную часть инварианта. Пусть b_i, a_{ij} и B обозначают значения перед вызовом процедуры PIVOT, а $\widehat{b}_i$ — значения, возвращаемые процедурой PIVOT.

Во-первых, заметим, что $\widehat{b}_e \geq 0$, поскольку $b_l \geq 0$ согласно инварианту цикла, $a_{le} > 0$ согласно строкам 6 и 9 процедуры SIMPLEX, а $\widehat{b}_e = b_l/a_{le}$ согласно строке 3 процедуры PIVOT.

Для остальных индексов $i \in B - \{l\}$ имеем

$$\begin{aligned} \widehat{b}_i &= b_i - a_{ie}\widehat{b}_e && \text{(согласно строке 9 процедуры PIVOT)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(согласно строке 3 процедуры PIVOT)} \end{aligned} \tag{29.76}$$

Необходимо рассмотреть два случая: $a_{ie} > 0$ и $a_{ie} \leq 0$. Если $a_{ie} > 0$, то, поскольку l выбирается так, что

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{для всех } i \in B , \tag{29.77}$$

мы имеем

$$\begin{aligned} \widehat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(согласно (29.76))} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(согласно (29.77))} \\ &= b_i - b_i \\ &= 0 , \end{aligned}$$

и, таким образом, $\widehat{b}_i \geq 0$. Если $a_{ie} \leq 0$, то, поскольку a_{le}, b_i и b_l неотрицательны, из уравнения (29.76) вытекает, что неотрицательным должно быть и значение $\widehat{b}_i$.

Теперь докажем, что это базисное решение является допустимым, т.е. что все переменные имеют неотрицательные значения. Небазисные переменные устанавливаются равными нулю и, следовательно, являются неотрицательными. Каждая базисная переменная x_i задается уравнением

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \ .$$

В базисном решении $\bar{x}_i = b_i$. Используя вторую часть инварианта цикла, можно сделать вывод, что все базисные переменные $\bar{x}_i$ неотрицательны.

Завершение. Цикл **while** может завершиться одним из двух способов. Если его завершение связано с выполнением условия в строке 3, то текущее базисное решение является допустимым и это решение возвращается строкой 17. Второй способ завершения связан с возвращением сообщения "задача неограниченная" строкой 11. В этом случае в каждой итерации цикла **for** (строки 5–8) при выполнении строки 6 оказывается, что $a_{ie} \leq 0$. Рассмотрим решение $\bar{x}$, определяемое следующим образом:

$$\bar{x}_i = \begin{cases} \infty \ , & \text{если } i = e \ , \\ 0 \ , & \text{если } i \in N - \{e\} \ , \\ b_i - \sum_{j \in N} a_{ij} \bar{x}_j \ , & \text{если } i \in B \ . \end{cases}$$

Покажем, что данное решение является допустимым, т.е. что все переменные являются неотрицательными. Небазисные переменные, отличные от $\bar{x}_e$, равны нулю, а значение $\bar{x}_e = \infty > 0$, следовательно, все небазисные переменные неотрицательны. Для каждой базисной переменной $\bar{x}_i$ имеем

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij} \bar{x}_j \\ &= b_i - a_{ie} \bar{x}_e \ . \end{aligned}$$

Из инварианта цикла вытекает, что $b_i \geq 0$, и мы имеем $a_{ie} \leq 0$ и $\bar{x}_e = \infty > 0$. Таким образом, $\bar{x}_i \geq 0$.

Теперь покажем, что целевое значение этого решения $\bar{x}$ является неограниченным. Из уравнения (29.42) целевое значение равно

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \bar{x}_j \\ &= v + c_e \bar{x}_e \ . \end{aligned}$$

Поскольку $c_e > 0$ (согласно строке 4 процедуры SIMPLEX) и $\bar{x}_e = \infty$, целевое значение бесконечно, а значит, задача линейного программирования неограниченная. ■

Остается показать, что процедура SIMPLEX завершается и что после ее завершения возвращаемое решение является оптимальным. Оптимальность будет рассмотрена в разделе 29.4. Сейчас же мы рассмотрим завершение процедуры.

Завершение

В приведенном в начале данного раздела примере каждая итерация симплекс-алгоритма увеличивала целевое значение, связанное с базисным решением. В упр. 29.3.2 предлагается показать, что ни одна итерация процедуры SIMPLEX не может уменьшить целевое значение, связанное с базисным решением. К сожалению, может оказаться, что итерация оставляет целевое значение неизменным. Это явление называется ***вырожденностью***, и сейчас мы рассмотрим его более подробно.

Целевое значение изменяется в строке 14 процедуры PIVOT в результате присваивания $\widehat{v} = v + c_e \widehat{b}_e$. Поскольку вызов процедуры PIVOT в процедуре SIMPLEX происходит только при $c_e > 0$, целевое значение может остаться неизменным (т.е. $\widehat{v} = v$) только в том случае, когда $\widehat{b}_e$ будет равно нулю. Это значение вычисляется как $\widehat{b}_e = b_l / a_{le}$ в строке 3 процедуры PIVOT. Поскольку процедура PIVOT вызывается только при $a_{le} \neq 0$, для того, чтобы $\widehat{b}_e$ было равно нулю и, как следствие, целевое значение осталось неизменным, должно выполняться условие $b_l = 0$.

Такая ситуация действительно может возникнуть. Рассмотрим следующую задачу линейного программирования:

$$\begin{aligned} z &= \phantom{8 -{}} x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= \phantom{8 - x_1 -{}} x_2 - x_3 \,. \end{aligned}$$

Предположим, что в качестве вводимой переменной выбрана переменная x_1, а в качестве выводимой — x_4. После замещения получим следующую задачу:

$$\begin{aligned} z &= 8 \phantom{{}- x_2} + x_3 - x_4 \\ x_1 &= 8 - x_2 \phantom{{}+ x_3} - x_4 \\ x_5 &= \phantom{8 -{}} x_2 - x_3 \,. \end{aligned}$$

В этой ситуации единственная возможность замещения — когда вводимой переменной является x_3, а выводимой переменной — x_5. Поскольку $b_5 = 0$, после замещения целевое значение 8 останется неизменным:

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= \phantom{8 -{}} x_2 \phantom{{}- x_4} - x_5 \,. \end{aligned}$$

Целевое значение осталось неизменным, но представление задачи изменилось. К счастью, если мы продолжим замещение, выбрав в качестве вводимой переменной x_2, а в качестве выводимой — x_1, целевое значение увеличится (до 16) и симплекс-алгоритм сможет продолжить свою работу.

Вырожденность является единственным возможным препятствием для окончания симплекс-алгоритма, так как может привести к явлению, именуемому ***зацикливанием***, когда канонические формы на двух разных итерациях одинаковы. Из-за вырожденности процедура SIMPLEX может выбирать последовательность замещающих операций, которые оставляют целевое значение неизменным, но при этом повторяют последовательность из одних и тех же канонических форм. Поскольку алгоритм SIMPLEX детерминированный, в случае зацикливания он бесконечно проходит по одной и той же последовательности канонических форм, никогда не заканчивая свою работу.

Зацикливание — единственная причина, по которой процедура SIMPLEX может не завершить свою работу. Чтобы показать это, сначала разработаем некоторую дополнительную технику.

На каждой итерации процедура SIMPLEX в дополнение к множествам N и B поддерживает A, b, c и v. Хотя для эффективной реализации симплекс-алгоритма требуется явная поддержка A, b, c и v, можно обойтись и без нее. Другими словами, множества базисных и небазисных переменных достаточно для того, чтобы единственным образом определить каноническую форму. Перед тем как доказать этот факт, докажем одну полезную алгебраическую лемму.

Лемма 29.3

Пусть I представляет собой множество индексов и пусть для каждого $j \in I$ α_j и β_j — действительные числа, а x_j — действительная переменная. Пусть также γ — некоторое действительное число. Предположим, что для любых x_j выполняется следующее условие:

$$\sum_{j \in I} \alpha_j x_j = \gamma + \sum_{j \in I} \beta_j x_j \ . \tag{29.78}$$

Тогда $\alpha_j = \beta_j$ для каждого $j \in I$, и $\gamma = 0$.

Доказательство. Поскольку уравнение (29.78) выполняется для любых значений x_j, можно выбрать для них определенные значения, чтобы сделать заключения об α, β и γ. Выбрав $x_j = 0$ для всех $j \in I$, можно сделать вывод, что $\gamma = 0$. Теперь выберем произвольный индекс $j \in I$ и зададим $x_j = 1$ и $x_k = 0$ для всех $k \neq j$. В таком случае должно выполняться равенство $\alpha_j = \beta_j$. Поскольку индекс j выбирался из множества I произвольным образом, можно заключить, что $\alpha_j = \beta_j$ для всех $j \in I$. ■

Конкретная задача линейного программирования может иметь много различных канонических форм; вспомним, что любая каноническая форма имеет то же самое множество допустимых и оптимальных решений, что и исходная задача. Теперь покажем, что каноническая форма любой задачи линейного программирования уникальным образом определяется множеством базисных переменных, т.е. что с заданным набором базисных переменных связана единственная каноническая форма (с однозначно определяемым множеством коэффициентов в правой части).

Лемма 29.4

Пусть (A, b, c) представляет собой задачу линейного программирования в стандартной форме. Задание множества базисных переменных B однозначно определяет соответствующую каноническую форму.

Доказательство. Будем проводить доказательство от противного. Предположим, что существуют две различные канонические формы с одинаковым множеством базисных переменных B. Эти канонические формы должны также иметь одинаковые множества небазисных переменных $N = \{1, 2, \ldots, n+m\} - B$. Запишем первую каноническую форму как

$$z = v + \sum_{j \in N} c_j x_j \tag{29.79}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{для } i \in B \ , \tag{29.80}$$

а вторую как

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.81}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B \ . \tag{29.82}$$

Рассмотрим систему уравнений, образованную путем вычитания каждого уравнения строки (29.82) из соответствующего уравнения строки (29.80). Полученная система имеет вид

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{для } i \in B$$

или, что эквивалентно,

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B \ .$$

Теперь для всех $i \in B$ применим лемму 29.3, где $\alpha_j = a_{ij}$, $\beta_j = a'_{ij}$, $\gamma = b_i - b'_i$ и $I = N$. Поскольку $\alpha_j = \beta_j$, имеем $a_{ij} = a'_{ij}$ для всех $j \in N$, а поскольку $\gamma = 0$, то $b_i = b'_i$. Таким образом, в этих двух канонических формах матрицы A и A' и векторы b и b' идентичны. С помощью аналогичных рассуждений в упр. 29.3.1 показано, что в этом случае также справедливо $c = c'$ и $v = v'$; следовательно, рассматриваемые канонические формы должны быть идентичны. ■

Теперь можно показать, что зацикливание — единственная возможная причина, по которой процедура SIMPLEX может не завершиться.

Лемма 29.5

Если процедура SIMPLEX не завершается не более чем за $\binom{n+m}{m}$ итераций, она зацикливается.

Доказательство. Согласно лемме 29.4 множество базисных переменных B однозначно определяет каноническую форму. Всего имеется $n + m$ переменных, а $|B| = m$, так что существует не более чем $\binom{n+m}{m}$ способов выбрать B. Следовательно, всего имеется не более чем $\binom{n+m}{m}$ различных канонических форм. Поэтому, если процедура SIMPLEX совершает более чем $\binom{n+m}{m}$ итераций, она должна зациклиться. ■

Зацикливание теоретически возможно, но встречается чрезвычайно редко. Его можно избежать путем несколько более аккуратного выбора вводимых и выводимых переменных. Один из способов состоит в подаче на вход слабого возмущения, что приводит к невозможности получить два решения с одинаковым целевым значением. Второй способ заключается в том, чтобы всегда выбирать переменную с наименьшим индексом. Эта стратегия известна как ***правило Бленда*** (Bland's rule). Мы не будем приводить доказательства, что эти стратегии позволяют избежать зацикливания.

Лемма 29.6

Если в строках 4 и 9 процедуры SIMPLEX всегда выполняется выбор переменной с наименьшим индексом, процедура SIMPLEX должна завершиться. ■

Мы завершим данный раздел следующей леммой.

Лемма 29.7

Если процедура INITIALIZE-SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым, то процедура SIMPLEX или выдает сообщение о неограниченности задачи линейного программирования, или завершается возвратом допустимого решения не более чем за $\binom{n+m}{m}$ итераций.

Доказательство. В леммах 29.2 и 29.6 показано, что если процедура INITIALIZE-SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым, то процедура SIMPLEX либо выдает сообщение о неограниченности задачи линейного программирования, либо завершается, предоставляя допустимое решение. Используя обращение леммы 29.5, приходим к заключению, что если процедура SIMPLEX завершается предоставлением допустимого решения, то это происходит не более чем за $\binom{n+m}{m}$ итераций. ■

Упражнения

29.3.1

Завершите доказательство леммы 29.4, показав, что $c = c'$ и $v = v'$.

29.3.2

Покажите, что значение v никогда не уменьшается в результате вызова процедуры PIVOT в строке 12 процедуры SIMPLEX.

29.3.3

Докажите, что каноническая форма, переданная процедуре PIVOT, эквивалентна возвращаемой ею.

29.3.4

Предположим, что мы преобразовали задачу линейного программирования (A, b, c) из стандартной формы в каноническую. Покажите, что базисное решение является допустимым тогда и только тогда, когда $b_i \geq 0$ для $i = 1, 2, \ldots, m$.

29.3.5

Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{array}{ll} \text{максимизировать} & 18x_1 + 12.5x_2 \\ \text{при условиях} & \\ & \begin{aligned} x_1 + x_2 &\leq 20 \\ x_1 &\leq 12 \\ x_2 &\leq 16 \\ x_1, x_2 &\geq 0. \end{aligned} \end{array}$$

29.3.6

Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{array}{ll} \text{максимизировать} & 5x_1 - 3x_2 \\ \text{при условиях} & \\ & \begin{aligned} x_1 - x_2 &\leq 1 \\ 2x_1 + x_2 &\leq 2 \\ x_1, x_2 &\geq 0. \end{aligned} \end{array}$$

29.3.7

Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{array}{ll} \text{минимизировать} & x_1 + x_2 + x_3 \\ \text{при условиях} & \\ & \begin{aligned} 2x_1 + 7.5x_2 + 3x_3 &\geq 10000 \\ 20x_1 + 5x_2 + 10x_3 &\geq 30000 \\ x_1, x_2, x_3 &\geq 0. \end{aligned} \end{array}$$

29.3.8

В доказательстве леммы 29.5 мы доказали, что имеется не более $\binom{m+n}{n}$ способов выбора множества B базисных переменных. Приведите пример задачи линейного

программирования, в которой количество способов выбора множества B базисных переменных строго меньше $\binom{m+n}{n}$.

29.4. Двойственность

Мы доказали, что при определенных предположениях процедура SIMPLEX завершается. Однако еще не доказано, что она действительно находит оптимальное решение задачи линейного программирования. Чтобы сделать это, введем новое мощное понятие — ***двойственность (дуальность) задач линейного программирования*** (linear-programming duality).

Двойственность позволяет доказать, что решение действительно оптимальное. С примерам двойственности мы встречались в главе 26, в теореме 26.6 о максимальном потоке и минимальном разрезе. Предположим, например, что в некоторой задаче поиска максимального потока мы нашли поток f величиной $|f|$. Как выяснить, является ли f максимальным потоком? Согласно теореме о максимальном потоке и минимальном разрезе, если можно найти разрез, значение которого также равно $|f|$, то тем самым подтверждается, что f действительно является максимальным потоком. Это пример двойственности: для задачи максимизации определяется связанная с ней задача минимизации, причем такая, что эти две задачи имеют одинаковые целевые значения.

Опишем, как для заданной задачи линейного программирования, в которой требуется максимизировать целевую функцию, сформулировать ***двойственную*** (dual) задачу линейного программирования, в которой целевую функцию требуется минимизировать и оптимальное значение которой идентично оптимальному значению исходной задачи. При работе с двойственными задачами исходная задача называется ***прямой*** (primal).

Для прямой задачи линейного программирования в стандартной форме, такой как (29.16)–(29.18), определим двойственную задачу следующим образом:

$$\text{минимизировать} \quad \sum_{i=1}^{m} b_i y_i \tag{29.83}$$

при условиях

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \text{ для } j = 1, 2, \ldots, n , \tag{29.84}$$

$$y_i \geq 0 \text{ для } i = 1, 2, \ldots, m . \tag{29.85}$$

Чтобы получить двойственную задачу, мы изменили максимизацию на минимизацию, поменяли роли коэффициентов правых частей и целевой функции и заменили неравенства “меньше или равно” неравенствами “больше или равно”. С каждым из m ограничений прямой задачи в двойственной задаче связана переменная y_i, а с каждым из n ограничений двойственной задачи связана переменная прямой задачи x_j. Например, рассмотрим задачу линейного программирования, заданную уравнениями (29.53)–(29.57). Двойственная ей задача имеет следую-

щий вид:

$$\begin{aligned} \text{минимизировать} \quad & 30y_1 + 24y_2 + 36y_3 & (29.86) \\ \text{при условиях} \quad & & \\ & y_1 + 2y_2 + 4y_3 \geq 3 & (29.87) \\ & y_1 + 2y_2 + y_3 \geq 1 & (29.88) \\ & 3y_1 + 5y_2 + 2y_3 \geq 2 & (29.89) \\ & y_1, y_2, y_3 \geq 0. & (29.90) \end{aligned}$$

В теореме 29.10 мы покажем, что оптимальное значение двойственной задачи линейного программирования всегда равно оптимальному значению прямой задачи. Более того, симплекс-алгоритм неявно решает одновременно обе задачи, прямую и двойственную, тем самым обеспечивая доказательство оптимальности.

Начнем с демонстрации ***слабой двойственности*** (weak duality), которая состоит в утверждении, что любое допустимое решение прямой задачи линейного программирования имеет целевое значение, не превышающее целевого значения любого допустимого решения двойственной задачи линейного программирования.

Лемма 29.8 (Слабая двойственность задач линейного программирования)

Пусть $\bar{x}$ представляет собой произвольное допустимое решение прямой задачи линейного программирования (29.16)–(29.18), а $\bar{y}$ — любое допустимое решение дуальной задачи линейного программирования (29.83)–(29.85). Тогда

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i \ .$$

Доказательство.

$$\begin{aligned} \sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j & \text{(согласно (29.84))} \\ &= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i & \\ &\leq \sum_{i=1}^{m} b_i \bar{y}_i & \text{(согласно (29.17))} \end{aligned}$$

■

Следствие 29.9

Пусть $\bar{x}$ является допустимым решением прямой задачи линейного программирования (A, b, c), а $\bar{y}$ — допустимым решением соответствующей двойственной задачи. Если

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i \ ,$$

то $\bar{x}$ и $\bar{y}$ являются оптимальными решениями прямой и двойственной задач соответственно.

Доказательство. Согласно лемме 29.8 целевое значение допустимого решения прямой задачи не превышает целевого значения допустимого решения двойственной задачи. Прямая задача является задачей максимизации, а двойственная — задачей минимизации. Поэтому, если допустимые решения $\bar{x}$ и $\bar{y}$ имеют одинаковые целевые значения, ни одно из них невозможно улучшить. ■

Прежде чем доказывать, что всегда существует решение двойственной задачи, целевое значение которого равно целевому значению оптимального решения прямой задачи, покажем, как найти такое решение. При решении задачи линейного программирования (29.53)–(29.57) с помощью симплекс-алгоритма последняя итерация дает каноническую форму (29.72)–(29.75) с целевым значением $z = 28 - x_3/6 - x_5/6 - 2x_6/3$, $B = \{1, 2, 4\}$ и $N = \{3, 5, 6\}$. Как будет показано ниже, базисное решение, связанное с этой последней канонической формой, является оптимальным решением задачи линейного программирования; таким образом, оптимальным решением задачи (29.53)–(29.57) является $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$ с целевым значением $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Как мы также покажем ниже, можно легко получить оптимальное решение двойственной задачи: оптимальные значения двойственных переменных противоположны коэффициентам целевой функции прямой задачи. Более строго, предположим, что последняя каноническая форма прямой задачи имеет вид

$$z = v' + \sum_{j \in N} c'_j x_j$$
$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B .$$

Тогда оптимальное решение двойственной задачи можно найти следующим образом:

$$\bar{y}_i = \begin{cases} -c'_{n+i} , & \text{если } (n+i) \in N , \\ 0 & \text{в противном случае} . \end{cases} \tag{29.91}$$

Таким образом, оптимальным решением двойственной задачи линейного программирования (29.86)–(29.90) является $\bar{y}_1 = 0$ (поскольку $n + 1 = 4 \in B$), $\bar{y}_2 = -c'_5 = 1/6$ и $\bar{y}_3 = -c'_6 = 2/3$. Вычисляя значение целевой функции двойственной задачи (29.86), получаем целевое значение $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$; это подтверждает, что целевое значение прямой задачи действительно равно целевому значению двойственной задачи. Используя эти вычисления и лемму 29.8, получаем доказательство, что оптимальное целевое значение прямой задачи линейного программирования равно 28. Теперь покажем, что этот подход применим в общем случае: таким образом можно найти оптимальное решение двойственной задачи и одновременно доказать оптимальность решения прямой задачи.

Теорема 29.10 (Двойственность задач линейного программирования)

Предположим, что процедура SIMPLEX возвращает значения $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ для прямой задачи линейного программирования (A, b, c). Пусть N и B — множества небазисных и базисных переменных окончательной канонической формы, c' — ее коэффициенты, а $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ определяется уравнением (29.91). Тогда $\bar{x}$ — оптимальное решение прямой задачи линейного программирования, $\bar{y}$ — оптимальное решение двойственной задачи и

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i \,. \tag{29.92}$$

Доказательство. Согласно следствию 29.9, если удастся найти допустимые решения $\bar{x}$ и $\bar{y}$, которые удовлетворяют уравнению (29.92), то $\bar{x}$ и $\bar{y}$ должны быть оптимальными решениями прямой и двойственной задач. Покажем, что решения $\bar{x}$ и $\bar{y}$, описанные в формулировке теоремы, удовлетворяют уравнению (29.92).

Предположим, что мы решаем прямую задачу линейного программирования (29.16)–(29.18) с помощью процедуры SIMPLEX. В ходе работы алгоритма строится последовательность канонических форм, пока он не завершится, предоставив окончательную каноническую форму с целевой функцией

$$z = v' + \sum_{j \in N} c'_j x_j \,. \tag{29.93}$$

Поскольку процедура SIMPLEX завершается с предоставлением решения, то согласно условию в строке 3 мы знаем, что

$$c'_j \le 0 \quad \text{для всех } j \in N \,. \tag{29.94}$$

Если мы определим

$$c'_j = 0 \quad \text{для всех } j \in B \,, \tag{29.95}$$

то уравнение (29.93) можно переписать как

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \qquad (\text{поскольку } c'_j = 0, \text{ если } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \qquad (\text{так как } N \cup B = \{1, 2, \ldots, n+m\}) \,. \end{aligned} \tag{29.96}$$

В базисном решении $\bar{x}$, связанном с конечной канонической формой, $\bar{x}_j = 0$ для всех $j \in N$, а $z = v'$. Поскольку все канонические формы эквивалентны, при вычислении значения исходной целевой функции для решения $\bar{x}$ мы должны

получить то же самое целевое значение:

$$\begin{aligned}\sum_{j=1}^{n} c_j\bar{x}_j &= v' + \sum_{j=1}^{n+m} c'_j\bar{x}_j \qquad (29.97)\\ &= v' + \sum_{j\in N} c'_j\bar{x}_j + \sum_{j\in B} c'_j\bar{x}_j\\ &= v' + \sum_{j\in N}(c'_j\cdot 0) + \sum_{j\in B}(0\cdot \bar{x}_j) \qquad (29.98)\\ &= v' .\end{aligned}$$

Теперь покажем, что решение $\bar{y}$, заданное формулой (29.91), является допустимым решением двойственной задачи и его целевое значение $\sum_{i=1}^{m} b_i\bar{y}_i$ равно $\sum_{j=1}^{n} c_j\bar{x}_j$. Из уравнения (29.97) следует, что значения целевых функций первой и последней канонических форм, вычисленные для $\bar{x}$, равны. В общем случае из эквивалентности всех канонических форм вытекает, что для *любого* набора значений $x = (x_1, x_2, \ldots, x_n)$ справедливо равенство

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j .$$

Следовательно, для любого набора значений $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ имеем

$$\begin{aligned}\sum_{j=1}^{n} c_j\bar{x}_j &= v' + \sum_{j=1}^{n+m} c'_j\bar{x}_j\\ &= v' + \sum_{j=1}^{n} c'_j\bar{x}_j + \sum_{j=n+1}^{n+m} c'_j\bar{x}_j\\ &= v' + \sum_{j=1}^{n} c'_j\bar{x}_j + \sum_{i=1}^{m} c'_{n+i}\bar{x}_{n+i}\\ &= v' + \sum_{j=1}^{n} c'_j\bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i)\,\bar{x}_{n+i} \qquad \text{(согласно (29.91) и (29.95))}\\ &= v' + \sum_{j=1}^{n} c'_j\bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i)\left(b_i - \sum_{j=1}^{n} a_{ij}\bar{x}_j\right) \qquad \text{(согласно (29.32))}\\ &= v' + \sum_{j=1}^{n} c'_j\bar{x}_j - \sum_{i=1}^{m} b_i\bar{y}_i + \sum_{i=1}^{m}\sum_{j=1}^{n} (a_{ij}\bar{x}_j)\,\bar{y}_i\end{aligned}$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i) \bar{x}_j$$

$$= \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i\right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i\right) \bar{x}_j ,$$

так что

$$\sum_{j=1}^{n} c_j \bar{x}_j = \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i\right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i\right) \bar{x}_j . \tag{29.99}$$

Применив лемму 29.3 к уравнению (29.99), получаем

$$v' - \sum_{i=1}^{m} b_i \bar{y}_i = 0 , \tag{29.100}$$

$$c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j \quad \text{для } j = 1, 2, \ldots, n . \tag{29.101}$$

Согласно уравнению (29.100) $\sum_{i=1}^{m} b_i \bar{y}_i = v'$, следовательно, целевое значение двойственной задачи $\left(\sum_{i=1}^{m} b_i \bar{y}_i\right)$ равно целевому значению прямой задачи (v'). Осталось показать, что $\bar{y}$ является допустимым решением двойственной задачи. Из (29.94) и (29.95) следует, что $c'_j \leq 0$ для всех $j = 1, 2, \ldots, n + m$. Поэтому для любого $j = 1, 2, \ldots, n$ из (29.101) следует, что

$$c_j = c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i$$
$$\leq \sum_{i=1}^{m} a_{ij} \bar{y}_i ,$$

что удовлетворяет ограничениям (29.84) двойственной задачи. Наконец, поскольку $c'_j \leq 0$ для всех $j \in N \cup B$, то, задав $\bar{y}$ в соответствии с уравнением (29.91), получим, что все $\bar{y}_i \geq 0$, так что ограничения неотрицательности также удовлетворены. ■

Итак, мы показали, что если данная задача линейного программирования разрешима, процедура INITIALIZE-SIMPLEX возвращает допустимое решение и процедура SIMPLEX завершается, не выдав сообщение "задача неограниченная", то возвращенное решение является оптимальным. Мы также показали, как строится оптимальное решение двойственной задачи линейного программирования.

Упражнения

29.4.1

Сформулируйте двойственную задачу для задачи, приведенной в упр. 29.3.5.

29.4.2

Предположим, имеется задача линейного программирования, не приведенная к стандартной форме. Можно получать двойственную задачу в два этапа: сначала привести исходную задачу к стандартной форме, а затем сформулировать двойственную. Однако было бы удобно иметь возможность сразу формулировать двойственную задачу. Объясните, каким образом, имея произвольную задачу линейного программирования, можно получить двойственную задачу непосредственно.

29.4.3

Запишите двойственную задачу для задачи поиска максимального потока, заданной уравнениями (29.47)–(29.50). Объясните, как интерпретировать эту формулировку в качестве задачи минимального разреза.

29.4.4

Запишите двойственную задачу для задачи потока с минимальной стоимостью, заданной уравнениями (29.51)–(29.52). Объясните, как интерпретировать полученную задачу в терминах графов и потоков.

29.4.5

Покажите, что задача, двойственная к двойственной задаче линейного программирования, является прямой задачей.

29.4.6

Какой результат из главы 26 можно интерпретировать как слабую двойственность для задачи поиска максимального потока?

29.5. Начальное базисное допустимое решение

В этом разделе мы покажем, как проверить, является ли задача линейного программирования разрешимой, и как в случае положительного ответа получить каноническую форму, базисное решение которой является допустимым. В заключение мы докажем основную теорему линейного программирования, в которой утверждается, что процедура SIMPLEX всегда дает правильный результат.

Поиск начального решения

В разделе 29.3 предполагалось, что у нас есть некая процедура INITIALIZE-SIMPLEX, которая определяет, имеет ли задача линейного программирования допустимые решения, и в случае положительного ответа возвращает каноническую форму, имеющую допустимое базисное решение. Опишем данную процедуру.

Даже если задача линейного программирования является разрешимой, начальное базисное решение может оказаться недопустимым. Рассмотрим, например,

следующую задачу линейного программирования:

максимизировать $2x_1 - x_2$ (29.102)

при условиях

$$2x_1 - x_2 \leq 2 \quad (29.103)$$
$$x_1 - 5x_2 \leq -4 \quad (29.104)$$
$$x_1, x_2 \geq 0. \quad (29.105)$$

Если преобразовать эту задачу в каноническую форму, базисным решением будет $x_1 = 0$, $x_2 = 0$. Это решение нарушает ограничение (29.104) и, следовательно, не является допустимым. Таким образом, процедура INITIALIZE-SIMPLEX не может сразу возвратить эту очевидную каноническую форму. Чтобы определить, существуют ли допустимые решения заданной задачи линейного программирования, можно сформулировать ***вспомогательную задачу линейного программирования*** (auxiliary linear program). Для этой вспомогательной задачи мы сможем (причем достаточно легко) найти каноническую форму, базисное решение которой является допустимым. Более того, решение этой вспомогательной задачи линейного программирования позволит определить, является ли исходная задача разрешимой, и, если является, обеспечит допустимое решение, которым можно инициализировать процедуру SIMPLEX.

Лемма 29.11

Пусть L представляет собой задачу линейного программирования в стандартной форме, заданную уравнениями (29.16)–(29.18), а L_{aux} — следующую задачу линейного программирования с $n + 1$ переменными:

максимизировать $-x_0$ (29.106)

при условиях

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i \text{ для } i = 1, 2, \ldots, m , \quad (29.107)$$
$$x_j \geq 0 \text{ для } j = 0, 1, \ldots, n . \quad (29.108)$$

Задача L разрешима тогда и только тогда, когда оптимальное целевое значение задачи L_{aux} равно нулю.

Доказательство. Предположим, что L имеет допустимое решение $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$. Тогда решение $\bar{x}_0 = 0$ в комбинации с $\bar{x}$ является допустимым решением задачи L_{aux} с целевым значением, равным 0. Поскольку в L_{aux} присутствует ограничение $x_0 \geq 0$, а цель состоит в максимизации $-x_0$, это решение должно быть оптимальным решением L_{aux}.

И обратно, предположим, что оптимальное целевое значение задачи L_{aux} равно нулю. Тогда $\bar{x}_0 = 0$, и значения остальных переменных $\bar{x}$ удовлетворяют ограничениям L. ■

Теперь опишем стратегию поиска начального базисного допустимого решения задачи линейного программирования L, заданной в стандартной форме:

INITIALIZE-SIMPLEX(A, b, c)

1 Пусть k является индексом минимального b_i
2 **if** $b_k \geq 0$ // Допустимо ли начальное базисное решение?
3 **return** $(\{1, 2, \ldots, n\}, \{n+1, n+2, \ldots, n+m\}, A, b, c, 0)$
4 Образуем L_{aux} путем добавления $-x_0$ к левой части каждого ограничения и задаем целевую функцию $-x_0$
5 Пусть (N, B, A, b, c, v) представляет собой результирующую каноническую форму для L_{aux}
6 $l = n + k$
7 // L_{aux} имеет $n + 1$ небазисную и m базисных переменных
8 $(N, B, A, b, c, v) =$ PIVOT$(N, B, A, b, c, v, l, 0)$
9 // Базисное решение является допустимым для L_{aux}
10 Выполняем итерации цикла **while** в строках 3–12 процедуры SIMPLEX, пока не будет найдено оптимальное решение задачи L_{aux}
11 **if** в оптимальном решении L_{aux} $\bar{x}_0 = 0$
12 **if** $\bar{x}_0$ является базисной переменной
13 выполнить одно (вырожденное) замещение, чтобы сделать ее небазисной
14 В окончательной канонической форме для L_{aux} удалить из ограничений x_0 и восстановить исходную целевую функцию L, но заменить в этой целевой функции каждую базисную переменную правой частью связанного с ней ограничения
15 **return** полученную окончательную каноническую форму
16 **else return** "задача неразрешима"

Процедура INITIALIZE-SIMPLEX работает следующим образом. В строках 1–3 неявно проверяется базисное решение исходной канонической формы задачи L, задаваемой $N = \{1, 2, \ldots, n\}$, $B = \{n+1, n+2, \ldots, n+m\}$, $\bar{x}_i = b_i$ для всех $i \in B$ и $\bar{x}_j = 0$ для всех $j \in N$. (Создание данной канонической формы не требует дополнительных усилий, поскольку значения A, b и c в стандартной и канонической формах одинаковы.) Если в строке 2 выясняется, что это базисное решение допустимо, т.е. $\bar{x}_i \geq 0$ для всех $i \in N \cup B$, то в строке 3 возвращается данная каноническая форма. В противном случае в строке 4 формируется вспомогательная задача линейного программирования L_{aux} так, как описано в лемме 29.11. Поскольку начальное базисное решение задачи L недопустимо, начальное базисное решение канонической формы L_{aux} также не может быть допустимым. Чтобы найти базисное допустимое решение, мы выполняем единственную операцию замещения. В строке 6 в качестве индекса базисной переменной, которая будет выводимой в этой операции замещения, выбирается $l = n + k$. Поскольку базисными переменными являются $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$, выводимая переменная x_l является переменной с наиболее отрицательным значением. В строке 8 выполняется соответствующий вызов процедуры PIVOT, в котором вводимой переменной является x_0, а выводимой — x_l. Немного позже мы покажем, что базисное реше-

30 Зак. 3726

ние, полученное в результате этого вызова процедуры PIVOT, будет допустимым. Имея каноническую форму, базисное решение которой является допустимым, мы можем (строка 10) многократно вызывать процедуру PIVOT до тех пор, пока не будет найдено окончательное решение вспомогательной задачи линейного программирования. Если проверка в строке 11 показывает, что найдено оптимальное решение задачи L_{aux} с целевым значением, равным нулю, то в строках 12–14 для задачи L создается каноническая форма, базисное решение которой является допустимым. Для этого сначала в строках 12–13 обрабатывается вырожденный случай, когда x_0 может оставаться базисной переменной со значением $\bar{x}_0 = 0$. В этом случае мы выполняем шаг замещения для удаления x_0 из базисных переменных, используя для вводимой переменной любое $e \in N$, такое, что $a_{0e} \neq 0$. Новое базисное решение остается допустимым; вырожденное замещение не изменяет значение ни одной из переменных. Затем из ограничений удаляются все члены с x_0 и восстанавливается исходная целевая функция задачи L. Исходная целевая функция может содержать как базисные, так и небазисные переменные. Поэтому все вхождения базисных переменных в целевой функции заменяются правыми частями соответствующих ограничений. Затем в строке 15 возвращается эта модифицированная каноническая форма. Если же в строке 11 обнаруживается, что исходная задача линейного программирования неразрешима, то сообщение об этом возвращается в строке 16.

Теперь покажем, как работает процедура INITIALIZE-SIMPLEX, на примере задачи линейного программирования (29.102)–(29.105). Эта задача будет разрешимой, если удастся найти неотрицательные значения x_1 и x_2, удовлетворяющие неравенствам (29.103) и (29.104). Используя лемму 29.11, сформулируем вспомогательную задачу линейного программирования:

максимизировать $-x_0$ (29.109)

при условиях

$$2x_1 - x_2 - x_0 \leq 2 \qquad (29.110)$$

$$x_1 - 5x_2 - x_0 \leq -4 \qquad (29.111)$$

$$x_1, x_2, x_0 \geq 0 .$$

Согласно лемме 29.11, если оптимальное целевое значение этой вспомогательной задачи равно нулю, исходная задача имеет допустимое решение. Если же оптимальное целевое значение вспомогательной задачи отрицательно, то исходная задача не имеет допустимых решений.

Запишем вспомогательную задачу в канонической форме:

$$\begin{aligned} z &= && - x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0 . \end{aligned}$$

Пока что базисное решение, в котором $x_4 = -4$, не является допустимым решением данной вспомогательной задачи. Однако с помощью единственного вызова процедуры PIVOT эту каноническую форму можно превратить в такую, базисное решение которой будет допустимым. Согласно строке 8 мы выбираем в качестве

вводимой переменной x_0. В строке 6 в качестве выводимой переменной выбирается x_4 — базисная переменная, которая в базисном решении имеет наибольшее по абсолютной величине отрицательное значение. После замещения получаем следующую каноническую форму:

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 \,. \end{aligned}$$

Связанное с ней базисное решение $(\bar{x}_0, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4) = (4, 0, 0, 6, 0)$ является допустимым. Теперь мы многократно вызываем процедуру PIVOT — до тех пор, пока не получим оптимальное решение задачи L_{aux}. В данном случае после одного вызова PIVOT с вводимой переменной x_2 и выводимой переменной x_0 получаем

$$\begin{aligned} z &= \phantom{\frac{14}{5}} - x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \,. \end{aligned}$$

Эта каноническая форма является окончательным решением вспомогательной задачи. Поскольку в данном решении $x_0 = 0$, можно сделать вывод, что исходная задача разрешима. Более того, поскольку $x_0 = 0$, эту переменную можно просто удалить из множества ограничений. Затем следует восстановить исходную целевую функцию, в которой сделаны соответствующие подстановки, чтобы она содержала только небазисные переменные. В нашем примере целевая функция приобретает вид

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right) \,.$$

Присвоив $x_0 = 0$ и упростив, получаем целевую функцию

$$-\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5}$$

и каноническую форму

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \,. \end{aligned}$$

Эта каноническая форма имеет допустимое базисное решение, и ее можно возвратить процедуре SIMPLEX.

Докажем формально корректность процедуры INITIALIZE-SIMPLEX.

Лемма 29.12

Если задача линейного программирования L не имеет допустимых решений, то процедура INITIALIZE-SIMPLEX возвращает сообщение "задача неразрешима". В противном случае процедура возвращает корректную каноническую форму, базисное решение которой является допустимым.

Доказательство. Сначала предположим, что задача линейного программирования L не имеет допустимых решений. Тогда согласно лемме 29.11 оптимальное целевое значение задачи L_{aux}, заданной формулами (29.106)–(29.108), является ненулевым, а поскольку x_0 подчиняется ограничению неотрицательности, оптимальное целевое значение должно иметь отрицательное целевое значение. Кроме того, это целевое значение должно быть конечным, так как решение, в котором $x_i = 0$ для $i = 1, 2, \ldots, n$ и $x_0 = |\min_{i=1}^{m} \{b_i\}|$, является допустимым и имеет целевое значение $-|\min_{i=1}^{m} \{b_i\}|$. Следовательно, в строке 10 процедуры INITIALIZE-SIMPLEX будет найдено решение с неположительным целевым значением. Пусть $\bar{x}$ — базисное решение, связанное с окончательной канонической формой. Мы не можем получить $\bar{x}_0 = 0$, поскольку тогда задача L_{aux} имела бы нулевое целевое значение, а это противоречит тому факту, что целевое значение отрицательно. Таким образом, проверка в строке 11 приведет к тому, что в строке 16 будет возвращено сообщение "задача неразрешима".

Теперь предположим, что задача линейного программирования L имеет допустимое решение. Из упр. 29.3.4 мы знаем, что если $b_i \geq 0$ для $i = 1, 2, \ldots, m$, то базисное решение, связанное с начальной канонической формой, является допустимым. В этом случае в строках 2 и 3 будет возвращена каноническая форма, связанная с исходной задачей. (Для преобразования стандартной формы в каноническую не требуется много усилий, поскольку A, b и c одинаковы и в той, и в другой формах.)

Для завершения доказательства рассмотрим случай, когда задача разрешима, но каноническая форма не возвращается в строке 3. Докажем, что в этом случае в строках 4–10 выполняется поиск допустимого решения задачи L_{aux} с нулевым целевым значением. Согласно строкам 1 и 2 в данном случае

$$b_k < 0$$

и

$$b_k \leq b_i \quad \text{для каждого } i \in B . \tag{29.112}$$

В строке 8 выполняется одна операция замещения, в процессе которой из базиса выводится переменная x_l (вспомним, что $l = n + k$, так что $b_l < 0$), стоящая в левой части уравнения с минимальным b_i, и вводится дополнительная переменная x_0. Покажем, что после такого замещения все элементы b являются неотрицательными и, следовательно, данное базисное решение задачи L_{aux} является допустимым. Обозначим через $\bar{x}$ базисное решение после вызова процедуры PIVOT, и пусть $\widehat{b}$ и $\widehat{B}$ представляют собой значения, возвращенные процедурой

PIVOT. Из леммы 29.1 следует, что

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\widehat{b}_e \ , & \text{если } i \in \widehat{B} - \{e\} \ , \\ b_l/a_{le} \ , & \text{если } i = e \ . \end{cases} \qquad (29.113)$$

При вызове процедуры PIVOT в строке 8 $e = 0$. Если переписать неравенства (29.107), чтобы включить коэффициенты a_{i0},

$$\sum_{j=0}^{n} a_{ij}x_j \leq b_i \quad \text{для } i = 1, 2, \ldots, m \ , \qquad (29.114)$$

то имеем

$$a_{i0} = a_{ie} = -1 \quad \text{для каждого } i \in B \ . \qquad (29.115)$$

(Заметим, что a_{i0} — это коэффициент при x_0 в выражении (29.114), а не противоположное ему значение, поскольку задача L_{aux} находится в стандартной, а не канонической форме.) Поскольку $l \in B$, мы также имеем $a_{le} = -1$. Таким образом, $b_l/a_{le} > 0$ и, следовательно, $\bar{x}_e > 0$. Для остальных базисных переменных получаем:

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\widehat{b}_e && \text{(согласно (29.113))} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(согласно строке 3 процедуры PIVOT)} \\ &= b_i - b_l && \text{(согласно (29.115) и } a_{le} = -1) \\ &\geq 0 && \text{(согласно (29.112))} \ , \end{aligned}$$

откуда вытекает, что теперь все базисные переменные неотрицательны. Следовательно, базисное решение, полученное в результате вызова процедуры PIVOT в строке 8, является допустимым. Следующей выполняется строка 10, которая решает задачу L_{aux}. Поскольку мы предположили, что задача L имеет допустимое решение, из леммы 29.11 следует, что задача L_{aux} имеет оптимальное решение с равным 0 целевым значением. Так как все канонические формы эквивалентны, в конечном базисном решении задачи L_{aux} должно выполняться $\bar{x}_0 = 0$, и после удаления из данной задачи переменной x_0 мы получим каноническую форму, базисное решение которой допустимо в задаче L. Эта каноническая форма возвращается в строке 15. ■

Основная теорема линейного программирования

Данная глава завершается демонстрацией корректности работы процедуры SIMPLEX. Произвольная задача линейного программирования или неразрешима, или неограничена, или имеет оптимальное решение с конечным целевым значением, и в каждом случае процедура SIMPLEX работает правильно.

Теорема 29.13 (Основная теорема линейного программирования)

Для любой задачи линейного программирования L, представленной в стандартной форме, возможен только один из следующих вариантов:

1. задача имеет оптимальное решение с конечным целевым значением;
2. задача является неразрешимой;
3. задача является неограниченной.

Если задача L является неразрешимой, процедура SIMPLEX возвращает сообщение "задача неразрешима". Если задача L является неограниченной, процедура SIMPLEX возвращает сообщение "задача неограниченная". В противном случае процедура SIMPLEX возвращает оптимальное решение с конечным целевым значением.

Доказательство. Согласно лемме 29.12, если задача линейного программирования L неразрешима, процедура SIMPLEX возвращает сообщение "задача неразрешима". Теперь предположим, что задача L разрешима. В соответствии с леммой 29.12 процедура INITIALIZE-SIMPLEX возвращает каноническую форму, базисное решение которой допустимо. Тогда согласно лемме 29.7 процедура SIMPLEX или возвращает сообщение "задача неограниченная", или завершается возвращением допустимого решения. Если она завершается и возвращает конечное решение, то это решение является оптимальным согласно теореме 29.10. Если же процедура SIMPLEX возвращает сообщение "задача неограниченная", то задача L является неограниченной согласно лемме 29.2. Поскольку процедура SIMPLEX всегда завершается одним из перечисленных способов, доказательство завершено. ∎

Упражнения

29.5.1

Напишите подробный псевдокод реализации строк 5 и 14 процедуры INITIALIZE-SIMPLEX.

29.5.2

Покажите, что при выполнении процедурой INITIALIZE-SIMPLEX основного цикла процедуры SIMPLEX никогда не возвращается сообщение "задача неограниченная".

29.5.3

Предположим, что дана задача линейного программирования L в стандартной форме и что для обеих задач, прямой и двойственной, базисные решения, связанные с начальными каноническими формами, являются допустимыми. Покажите, что оптимальное целевое значение задачи L равно 0.

29.5.4

Предположим, что в задачу линейного программирования разрешено включать строгие неравенства. Покажите, что в этом случае основная теорема линейного программирования не выполняется.

29.5.5

Решите следующую задачу линейного программирования с помощью процедуры SIMPLEX:

максимизировать $x_1 + 3x_2$
при условиях

$$\begin{aligned} x_1 - x_2 &\le 8 \\ -x_1 - x_2 &\le -3 \\ -x_1 + 4x_2 &\le 2 \\ x_1, x_2 &\ge 0. \end{aligned}$$

29.5.6

Решите следующую задачу линейного программирования с помощью процедуры SIMPLEX:

максимизировать $x_1 - 2x_2$
при условиях

$$\begin{aligned} x_1 + 2x_2 &\le 4 \\ -2x_1 - 6x_2 &\le -12 \\ x_2 &\le 1 \\ x_1, x_2 &\ge 0. \end{aligned}$$

29.5.7

Решите следующую задачу линейного программирования с помощью процедуры SIMPLEX:

максимизировать $x_1 + 3x_2$
при условиях

$$\begin{aligned} -x_1 + x_2 &\le -1 \\ -x_1 - x_2 &\le -3 \\ -x_1 + 4x_2 &\le 2 \\ x_1, x_2 &\ge 0. \end{aligned}$$

29.5.8

Решите задачу линейного программирования (29.6)–(29.10).

29.5.9

Рассмотрим задачу линейного программирования P с одной переменной

максимизировать tx
при условиях

$$\begin{aligned} rx &\le s \\ x &\ge 0, \end{aligned}$$

где r, s и t представляют собой произвольный действительные числа. Пусть D — двойственная к P задача.

При каких значениях r, s, и t можно утверждать, что:

1. обе задачи, P и D, имеют оптимальные решения с конечными целевыми значениями;
2. P является разрешимой, а D — неразрешимой;
3. D является разрешимой, а P — неразрешимой;
4. ни одна из задач P и D не является разрешимой.

Задачи

29.1. Разрешимость линейных неравенств

Пусть задано множество m линейных неравенств с n переменными $x_1, x_2, \ldots, x_n$. В ***задаче о разрешимости системы линейных неравенств*** требуется ответить, существует ли набор значений переменных, удовлетворяющий одновременно всем неравенствам.

а. Покажите, что для решения задачи о разрешимости системы линейных неравенств можно использовать алгоритм решения задачи линейного программирования. Число переменных и ограничений, используемых в задаче линейного программирования, должно полиномиально зависеть от n и m.

б. Покажите, что алгоритм решения задачи о разрешимости системы линейных неравенств можно использовать для решения задачи линейного программирования. Число переменных и линейных неравенств, используемых в задаче о разрешимости системы линейных неравенств, должно полиномиально зависеть от числа переменных n и числа ограничений m задачи линейного программирования.

29.2. Дополняющая нежесткость

Дополняющая нежесткость (complementary slackness) характеризует связь между значениями переменных прямой задачи и ограничениями двойственной задачи, а также между значениями переменных двойственной задачи и ограничениями прямой задачи. Пусть $\bar{x}$ — допустимое решение прямой задачи линейного программирования (29.16)–(29.18), а $\bar{y}$ — допустимое решение двойственной задачи (29.83)–(29.85). Принцип дополняющей нежесткости гласит, что следующие условия являются необходимыми и достаточными условиями оптимальности $\bar{x}$ и $\bar{y}$:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ или } \bar{x}_j = 0 \quad \text{для } j = 1, 2, \ldots, n$$

и

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \text{ или } \bar{y}_i = 0 \quad \text{для } i = 1, 2, \ldots, m .$$

а. Убедитесь, что принцип дополняющей нежесткости справедлив для задачи линейного программирования (29.53)–(29.57).

б. Докажите, что принцип дополняющей нежесткости справедлив для любой прямой и соответствующей ей двойственной задачи.

в. Докажите, что допустимое решение $\bar{x}$ прямой задачи линейного программирования (29.16)–(29.18) является оптимальным тогда и только тогда, когда существуют значения $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$, такие, что

1. $\bar{y}$ является допустимым решением двойственной задачи линейного программирования (29.83)–(29.85);
2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ для всех j, таких, что $\bar{x}_j > 0$;
3. $\bar{y}_i = 0$ для всех i, таких, что $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

29.3. Целочисленное линейное программирование

Задача целочисленного линейного программирования представляет собой задачу линейного программирования с дополнительным ограничением, состоящим в том, что переменные x должны принимать целые значения. В упр. 34.5.3 показано, что даже определение того, имеет ли задача целочисленного линейного программирования допустимые решения, является NP-сложной задачей; это значит, что вряд ли существует полиномиальный по времени алгоритм решения данной задачи.

а. Покажите, что для задач целочисленного линейного программирования справедлива слабая двойственность (лемма 29.8).

б. Покажите, что двойственность (теорема 29.10) не всегда присуща задачам целочисленного линейного программирования.

в. Пусть задана прямая задача линейного программирования в стандартной форме и пусть P — оптимальное целевое значение прямой задачи, D — оптимальное целевое значение ее двойственной задачи, IP — оптимальное целевое значение целочисленной версии прямой задачи (т.е. прямой задачи с дополнительным ограничением, состоящим в том, что переменные принимают только целые значения), а ID — оптимальное целевое значение целочисленной версии двойственной задачи. Исходя из предположения о том, что и прямая, и двойственная целочисленные задачи разрешимы и ограниченны, покажите, что

$$IP \leq P = D \leq ID .$$

29.4. Лемма Фаркаша

Пусть A — матрица размером $m \times n$, а c — n-мерный вектор. Лемма Фаркаша (Farkas) гласит, что разрешима только одна из систем

$$Ax \leq 0$$
$$c^{\mathrm{T}}x > 0$$

и

$$\begin{aligned} A^{\mathrm{T}} y &= c \\ y &\geq 0 , \end{aligned}$$

где x — n-мерный вектор, а y — m-мерный вектор. Докажите эту лемму.

29.5. Циркуляция с минимальной стоимостью

В этой задаче рассматривается вариант задачи потока минимальной стоимости из раздела 29.2, в которой не заданы ни целевое значение потока, ни исток и сток. Как и ранее, имеются транспортная сеть и стоимости ребер $a(u, v)$. Поток допусти́м, если он удовлетворяет ограничению пропускной способности в каждом ребре и сохранению потока в *каждой* вершине. Цель заключается в поиске среди всех допустимых потоков того, который обладает минимальной стоимостью. Эта задача называется задачей поиска циркуляции минимальной стоимости.

а. Сформулируйте задачу поиска циркуляции минимальной стоимости в виде задачи линейного программирования.

б. Предположим, что для всех ребер $(u, v) \in E$ значения стоимости $a(u, v) > 0$. Опишите оптимальное решение задачи поиска циркуляции минимальной стоимости.

в. Сформулируйте задачу поиска максимального потока в виде задачи линейного программирования для задачи поиска циркуляции минимальной стоимости. То есть для заданного экземпляра задачи о максимальном потоке $G = (V, E)$ с истоком s, стоком t и пропускными способностями ребер c разработайте задачу поиска циркуляции минимальной стоимости, задав (возможно, другую) сеть $G' = (V', E')$ с пропускными способностями ребер c' и стоимостями ребер a', такую, что можно получить решение задачи о максимальном потоке из решения задачи поиска циркуляции минимальной стоимости.

г. Сформулируйте задачу поиска кратчайшего пути из одного источника в виде задачи линейного программирования для задачи поиска циркуляции минимальной стоимости.

Заключительные замечания

В данной главе мы только приступили к изучению обширной области линейного программирования. Множество авторов написали книги, посвященные исключительно линейному программированию. Среди них Чватал (Chvátal) [68], Гасс (Gass) [129], Карлов (Karloff) [196], Шрайвер (Schrijver) [301] и Вандербей (Vanderbei) [342]. Во многих других книгах линейное программирование подробно рассматривается наряду с другими вопросами (см., например, работы Пападимитриу (Papadimitriou) и Штейглиц (Steiglitz) [269] и Ахуя (Ahuja), Магнанти

(Magnanti) и Орлин (Orlin) [7]). Изложение в данной главе построено на подходе, предложенном Чваталом (Chvátal).

Симплекс-алгоритм для задач линейного программирования был открыт Дж. Данцигом (G. Dantzig) в 1947 году. Немного позже было обнаружено, что множество задач из различных областей можно сформулировать в виде задач линейного программирования и решить с помощью симплекс-алгоритма. Осознание этого факта привело к стремительному росту использования линейного программирования и развитию его алгоритмов. Различные варианты симплекс-алгоритма остаются наиболее популярными методами решения задач линейного программирования. Эта история описана во многих работах, например в примечаниях к [68] и [196].

Эллипсоидный алгоритм, предложенный Л.Г. Хачияном в 1979 году, стал первым алгоритмом решения задач линейного программирования с полиномиальным временем выполнения. Он был основан на более ранних работах Н.З. Шора, Д.Б. Юдина и А.С. Немировского. Использование эллипсоидного алгоритма для решения разнообразных задач комбинаторной оптимизации описано в работе Грётшеля (Grötschel), Ловаса (Lovasz) и Шрайвера (Schrijver) [153]. Однако на сегодняшний день эллипсоидный алгоритм с точки зрения практического применения не может конкурировать с симплекс-алгоритмом.

В работе Кармаркара (Karmarkar) [197] содержится описание предложенного им алгоритма внутренней точки. Многие его последователи разработали свои алгоритмы внутренней точки. Хороший обзор этих алгоритмов можно найти в статье Гольдфарба (Goldfarb) и Тодда (Todd) [140] и в книге Ие (Ye) [359].

Анализ симплекс-алгоритма относится к области активных исследований. Кли (Klee) и Минти (Minty) построили пример, в котором симплекс-алгоритм выполняет $2^n - 1$ итераций. На практике симплекс-алгоритм работает очень эффективно, и многие исследователи пытались найти теоретическое обоснование этому эмпирическому наблюдению. Исследования, начатые Боргвардтом (Borgwardt) и продолженные многими другими, показывают, что при определенных вероятностных предположениях об исходных данных симплекс-алгоритм сходится за ожидаемое полиномиальное время. Последних результатов в этой области добились Спилмен (Spielman) и Тенг (Teng) [320], которые ввели понятие "сглаженный анализ алгоритмов" и применили его к симплекс-алгоритму.

Известно, что симплекс-алгоритм работает более эффективно в некоторых частных случаях. К примеру, заслуживает внимания сетевой симплекс-алгоритм — разновидность симплекс-алгоритма, приспособленная для решения задач сетевых потоков. Для некоторых сетевых задач, включая задачи поиска кратчайшего пути, максимального потока и потока минимальной стоимости, варианты сетевого симплекс-алгоритма достигают результата за полиномиальное время. Обратитесь, например, к статье Орлина (Orlin) [266] и ссылкам в ней.

Глава 30. Полиномы и быстрое преобразование Фурье

Для непосредственного сложения двух полиномов степени n требуется время $\Theta(n)$, однако для их непосредственного умножения требуется время $\Theta(n^2)$. В данной главе показано, как с помощью быстрого преобразования Фурье (БПФ) (Fast Fourier Transform — FFT) можно сократить время умножения полиномов до $\Theta(n \lg n)$.

Наиболее часто преобразования Фурье (а следовательно, и БПФ) используются в обработке сигналов. Сигнал задается во ***временной области*** (time domain) как функция, отображающая время в амплитуду. Анализ Фурье позволяет выразить сигнал как взвешенную сумму сдвинутых по фазе синусоид различных частот. Веса и фазы связаны с частотными характеристиками сигнала в ***частотной области*** (frequency domain). Множество современных приложений БПФ включает технологии сжатия цифровой видео- и аудиоинформации, в том числе MP3-файлы. Обработка сигналов — обширная область исследований, которой посвящено несколько отличных книг (в конце данной главы приводятся ссылки на некоторые из них).

Полиномы

Полиномом (polynomial) относительно переменной x над алгебраическим полем F называется представление функции $A(x)$ в виде формальной суммы

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

Значения $a_1, a_2, \ldots, a_{n-1}$ называются ***коэффициентами*** данного полинома. Коэффициенты выбираются из некоторого поля F; как правило, это множество комплексных чисел $\mathbb{C}$. Говорят, что полином $A(x)$ имеет ***степень*** k, если его старшим ненулевым коэффициентом является a_k; это записывается как $\text{degree}(A) = k$. Любое целое число, строго большее степени полинома, называется ***границей степени*** (degree-bound) данного полинома. Следовательно, степенью полинома с границей степени n может быть любое целое число от 0 до $n-1$ включительно.

Для полиномов можно определить множество разнообразных операций. Например, ***сложение полиномов*** (polynomial addition): если $A(x)$ и $B(x)$ — полиномы с границей степени n, то их ***суммой*** является полином $C(x)$, граница степени

которого также равна n, такой, что $C(x) = A(x) + B(x)$ для всех x из соответствующего поля. Таким образом, если

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

и

$$B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

то

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

где $c_j = a_j + b_j$ для $j = 0, 1, \ldots, n-1$. Например, если мы имеем полиномы $A(x) = 6x^3 + 7x^2 - 10x + 9$ и $B(x) = -2x^3 + 4x - 5$, то $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Умножение полиномов (polynomial multiplication) определяется следующим образом: если $A(x)$ и $B(x)$ — полиномы с границей степени n, то их ***произведением*** (product) $C(x)$ является полином с границей степени $2n - 1$, такой, что $C(x) = A(x)B(x)$ для всех x из соответствующего поля. Вероятно, вам уже приходилось перемножать полиномы, умножая каждый член полинома $A(x)$ на каждый член полинома $B(x)$ и выполняя объединение членов с одинаковыми степенями. Например, умножение полиномов $A(x) = 6x^3 + 7x^2 - 10x + 9$ и $B(x) = -2x^3 + 4x - 5$ можно выполнить следующим образом.

$$\begin{array}{r}
6x^3 + 7x^2 - 10x + 9 \\
-2x^3 + 4x - 5 \\
\hline
-30x^3 - 35x^2 + 50x - 45 \\
24x^4 + 28x^3 - 40x^2 + 36x \\
-12x^6 - 14x^5 + 20x^4 - 18x^3 \\
\hline
-12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
\end{array}$$

По-другому произведение $C(x)$ можно записать как

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j , \tag{30.1}$$

где

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} . \tag{30.2}$$

Заметим, что $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$, откуда вытекает, что если A — полином степени с границей степени n_a, а B — полином с границей степени n_b, то C — полином с границей степени $n_a + n_b - 1$. Тем не менее, поскольку полином с границей степени k является также полиномом с границей степени $k + 1$,

обычно мы будем говорить, что произведение полиномов C является полиномом с границей степени $n_a + n_b$.

Краткое содержание главы

В разделе 30.1 описаны два способа представления полиномов: представление, основанное на коэффициентах, и представление, основанное на значениях в точках. Для непосредственного умножения полиномов (уравнения (30.1) и (30.2)) требуется время $\Theta(n^2)$, если полиномы представлены с помощью коэффициентов, и только $\Theta(n)$, когда они представлены в форме, основанной на значениях в точках. Однако с помощью преобразования представлений можно умножить заданные с помощью коэффициентов полиномы за время $\Theta(n \lg n)$. Чтобы понять, как это происходит, необходимо сначала изучить свойства комплексных корней из единицы, что и предлагается сделать в разделе 30.2. Затем также описанные в разделе 30.2 прямое и обратное БПФ используются для выполнения указанных преобразований представлений. В разделе 30.3 показано, как быстро реализовать БПФ в последовательных и параллельных моделях.

В данной главе широко используются комплексные числа, поэтому символ i будет использоваться в ней исключительно для обозначения $\sqrt{-1}$.

30.1. Представление полиномов

Представления полиномов в форме коэффициентов и в форме значений в точках в определенном смысле эквивалентны: полиному, заданному в форме точек-значений, соответствует единственный полином в форме коэффициентов. В данном разделе мы познакомимся с обоими представлениями и покажем, как их можно скомбинировать, чтобы выполнить умножение двух полиномов с границей степени n за время $\Theta(n \lg n)$.

Представление, основанное на коэффициентах

Основанным на коэффициентах представлением (coefficient representation) полинома $A(x) = \sum_{j=0}^{n-1} a_j x^j$ с границей степени n является вектор коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$. В матричных уравнениях данной главы мы будем считать векторы векторами-столбцами.

Основанное на коэффициентах представление удобно при выполнении определенных операций над полиномами. Например, операция ***вычисления*** (evaluating) полинома $A(x)$ в некой заданной точке x_0 заключается в вычислении значения $A(x_0)$. Если использовать ***схему Горнера*** (Horner's rule), вычисление требует времени $\Theta(n)$:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)) .$$

Аналогично сложение двух полиномов, представленных векторами коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$ и $b = (b_0, b_1, \ldots, b_{n-1})$, занимает время $\Theta(n)$: мы просто создаем вектор коэффициентов $c = (c_0, c_1, \ldots, c_{n-1})$, где $c_j = a_j + b_j$ для $j = 0, 1, \ldots, n-1$.

Рассмотрим теперь умножение двух полиномов, $A(x)$ и $B(x)$, с границей степени n, представленных в форме коэффициентов. Если использовать метод, описанный уравнениями (30.1) и (30.2), умножение данных полиномов займет время $\Theta(n^2)$, поскольку каждый коэффициент из вектора a необходимо умножить на каждый коэффициент из вектора b. Операция умножения полиномов в форме коэффициентов гораздо сложнее, чем операции вычисления полинома или сложения двух полиномов. Результирующий вектор коэффициентов c, заданный формулой (30.2), также называется ***сверткой*** (convolution) исходных векторов a и b, и обозначается как $c = a \otimes b$. Поскольку умножение полиномов и вычисление сверток являются фундаментальными вычислительными задачами, имеющими важное практическое значение, данная глава посвящена эффективным алгоритмам их решения.

Представление, основанное на значениях в точках

Основанным на значениях в точках представлением (point-value representation) полинома $A(x)$ с границей степени n является множество, состоящее из n ***пар "точка–значение"*** (point-value pairs):

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

таких, что все x_k различны и

$$y_k = A(x_k) \tag{30.3}$$

для $k = 0, 1, \ldots, n-1$. Каждый полином имеет множество различных представлений, основанных на значениях в точках, поскольку в качестве базиса такого представления можно использовать любое множество n различных точек $x_0, x_1, \ldots, x_{n-1}$.

Получить основанное на значениях в точках представление полинома, заданного с помощью коэффициентов, достаточно просто: следует лишь выбрать n различных точек $x_0, x_1, \ldots, x_{n-1}$ и вычислить $A(x_k)$ для $k = 0, 1, \ldots, n-1$. С помощью схемы Горнера такое вычисление можно выполнить за время $\Theta(n^2)$. Далее мы покажем, что при разумном выборе x_k данное вычисление можно ускорить, и оно будет выполняться за время $\Theta(n \lg n)$.

Обратная процедура — определение коэффициентов полинома, заданного в форме значений в точках — называется ***интерполяцией*** (interpolation). В следующей теореме утверждается, что интерполяция является вполне определенной, когда граница степени искомого интерполяционного полинома равна числу заданных пар "точка–значение".

Теорема 30.1 (Единственность интерполяционного полинома)

Для любого множества $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, состоящего из n пар "точка–значение", таких, что все значения x_k различны, существует единственный полином $A(x)$ с границей степени n, такой, что $y_k = A(x_k)$ для $k = 0, 1, \ldots, n-1$.

Доказательство. Доказательство основано на существовании матрицы, обратной заданной. Уравнение (30.3) эквивалентно матричному уравнению

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} . \tag{30.4}$$

Матрица в левой части обозначается как $V(x_0, x_1, \ldots, x_{n-1})$ и называется матрицей Вандермонда (Vandermonde). Согласно упр. Г.1 определитель данной матрицы равен

$$\prod_{0 \le j < k \le n-1} (x_k - x_j) ,$$

следовательно, по теореме Г.5 она является обратимой (т.е. невырожденной), если все x_k различны. Таким образом, для заданного представления в виде значений в точках можно однозначно вычислить коэффициенты a_j:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y .$$

■

Доказательство теоремы 30.1 предлагает алгоритм интерполяции, основанный на решении системы линейных уравнений (30.4). Используя описанные в главе 28 алгоритмы LU-разложения, эти уравнения можно решить за время $O(n^3)$.

Более быстрый алгоритм n-точечной интерполяции основан на ***формуле Лагранжа*** (Lagrange's formula):

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} (x - x_j)}{\prod_{j \ne k} (x_k - x_j)} . \tag{30.5}$$

Можете самостоятельно убедиться в том, что правая часть уравнения (30.5) является полиномом степени не выше n, удовлетворяющим условию $y_k = A(x_k)$ для всех k. В упр. 30.1.5 предлагается показать, как с помощью формулы Лагранжа вычислить коэффициенты полинома A за время $\Theta(n^2)$.

Таким образом, вычисление и интерполяция по n точкам являются вполне определенными обратимыми операциями, которые позволяют преобразовать представление полинома в виде коэффициентов в представление в виде точек-

значений и обратно[1]. Решение этих задач с помощью описанных выше алгоритмов занимает время $\Theta(n^2)$.

Представление в виде значений в точках весьма удобно при выполнении многих операций над полиномами. При сложении, если $C(x) = A(x) + B(x)$, то $C(x_k) = A(x_k) + B(x_k)$ для любой точки x_k. Говоря более строго, если у нас есть основанное на значениях в точках представление полиномов A

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

и B

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(обратите внимание, что A и B вычисляются в *одних и тех же* n точках), то основанное на значениях в точках представление полинома C имеет вид

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Таким образом, время, необходимое для сложения двух полиномов с границей степени n, заданных в форме значений в точках, составляет $\Theta(n)$.

Точно так же представление в виде значений в точках удобно при выполнении умножения полиномов. Если $C(x) = A(x)B(x)$, то $C(x_k) = A(x_k)B(x_k)$ для любой точки x_k, и можно поточечно умножить представление A на соответствующее представление B и получить основанное на значениях в точках представление C. Однако возникает следующая проблема: $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$; если A и B имеют границу степени n, то граница степени результирующего полинома составляет $2n$. Стандартное представление каждого из полиномов A и B в виде значений в точках содержит n пар "точка–значение". В результате их перемножения получится n пар "точка–значение", однако для однозначной интерполяции полинома C с границей степени $2n$ требуется $2n$ пар (см. упр. 30.1.4.) Следовательно, необходимо использовать "расширенные" представления полиномов A и B, которые содержат по $2n$ пар "точка–значение". Если задано расширенное представление в виде значений в точках полинома A

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

и соответствующее расширенное представление в виде значений в точках полинома B

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} ,$$

то представление полинома C имеет вид

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

---

[1]Известно, что интерполяция является неприятной задачей с точки зрения численной устойчивости: хотя описанные здесь подходы математически корректны, небольшие различия во вводимых величинах или ошибкн округления в ходе вычислений могут привести к значительно различающимся результатам.

Если два исходных полинома заданы в расширенной форме "точки–значения", то время их умножения для получения результата в той же форме составляет $\Theta(n)$, что значительно меньше, чем время, необходимое для умножения полиномов, заданных коэффициентами.

Наконец рассмотрим, как вычислить значение полинома, заданного в виде значений в точках, в некоторой новой точке. По-видимому, для этой задачи не существует более простого подхода, чем преобразовать полином в форму коэффициентов, а затем вычислить его значение в новой точке.

Быстрое умножение полиномов, заданных коэффициентами

Можно ли использовать метод умножения полиномов, заданных в виде значений в точках, время выполнения которого линейно зависит от n, для ускорения умножения полиномов, заданных в форме коэффициентов? Ответ зависит от способности быстро выполнять преобразование полинома из формы коэффициентов в форму "точки–значения" (вычисление) и обратно (интерполяция).

В качестве точек вычисления можно использовать любые точки, однако тщательный выбор точек дает возможность выполнять переход от одного представления к другому за время $\Theta(n \lg n)$. Как будет показано в разделе 30.2, если в качестве точек вычисления выбрать комплексные корни из единицы, то представление "точки–значения" можно получить с помощью дискретного преобразования Фурье (ДПФ) (Discrete Fourier Transform — DFT) вектора коэффициентов. Обратную операцию, интерполяцию, можно выполнить путем применения обратного ДПФ к парам "точки–значения", в результате чего получается вектор коэффициентов. В разделе 30.2 будет показано, как БПФ позволяет выполнять прямое и обратное ДПФ за время $\Theta(n \lg n)$.

На рис. 30.1 данная стратегия представлена графически. Небольшая сложность связана с границами степеней. Произведение двух полиномов с границей степени n является полиномом с границей степени $2n$. Поэтому перед вычислением исходных полиномов A и B их границы степени удваиваются до $2n$ путем добавления n старших коэффициентов, равных нулю. Поскольку теперь векторы коэффициентов содержат по $2n$ элементов, мы используем комплексные корни $2n$-й степени из единицы, которые обозначены на рис. 30.1 как ω_{2n}.

Можно предложить следующую основанную на БПФ процедуру умножения полиномов $A(x)$ и $B(x)$ с границей степени n, в которой исходные полиномы и результат представлены в форме коэффициентов, а время выполнения составляет $\Theta(n \lg n)$. Предполагается, что n является степенью 2; это требование всегда можно удовлетворить, добавив равные нулю старшие коэффициенты.

1. *Удвоение границы степени.* Создаются представления в форме коэффициентов полиномов $A(x)$ и $B(x)$ в виде полиномов с границей степени $2n$ путем добавления n нулевых старших коэффициентов в каждое.

2. *Вычисление.* Определяются представления полиномов $A(x)$ и $B(x)$ в форме "точки–значения" длиной $2n$ путем двукратного применения БПФ порядка $2n$. Эти представления содержат значения двух заданных полиномов в точках, являющихся комплексными корнями степени $2n$ из единицы.

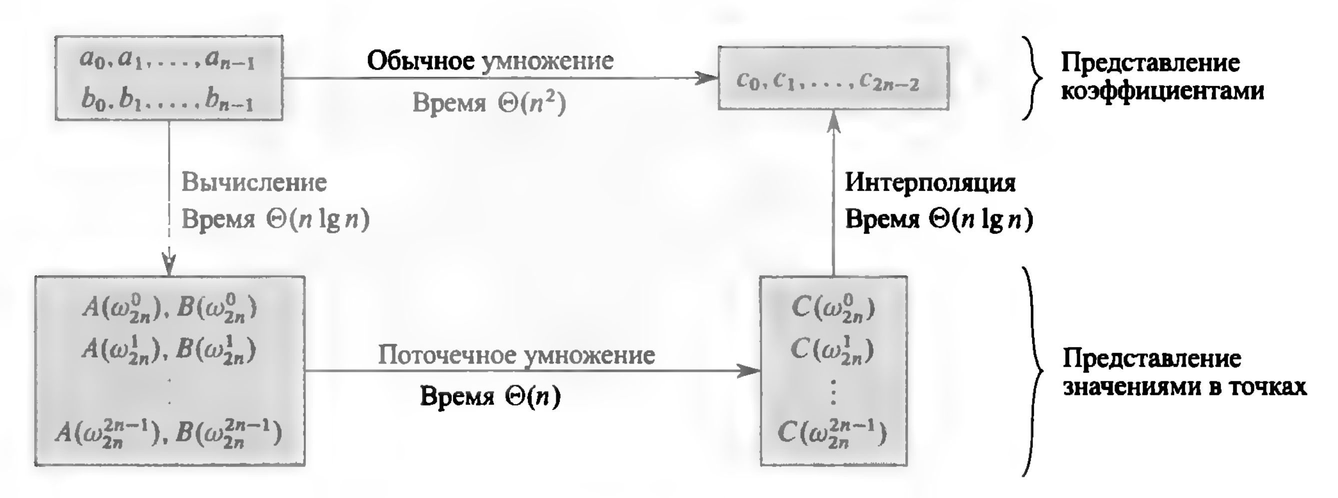


Рис. 30.1. Графическое представление эффективной процедуры умножения полиномов. Вверху показано представление в форме коэффициентов, а внизу — в форме значений в точках. Идущие слева направо стрелки соответствуют операции умножения. Члены ω_{2n} являются комплексными корнями степени $2n$ из единицы.

3. *Поточечное умножение.* Вычисляется представление в виде значений в точках полинома $C(x) = A(x)B(x)$ путем поточечного умножения соответствующих значений. Это представление содержит значения полинома $C(x)$ в каждом корне степени $2n$ из единицы.
4. *Интерполяция.* Создается представление полинома $C(x)$ ф форме коэффициентов с помощью однократного применения БПФ к $2n$ парам "точка–значение" для вычисления обратного ДПФ.

Этапы 1 и 3 выполняются за время $\Theta(n)$, а этапы 2 и 4 — за время $\Theta(n \lg n)$. Таким образом, показав, как использовать БПФ, мы докажем следующую теорему.

Теорема 30.2

Произведение двух полиномов с границей степени n в случае, когда исходные полиномы и результат находятся в форме коэффициентов, можно вычислить за время $\Theta(n \lg n)$. ■

Упражнения

30.1.1

Перемножьте полиномы $A(x) = 7x^3 - x^2 + x - 10$ и $B(x) = 8x^3 - 6x + 3$ с использованием (30.1) и (30.2).

30.1.2

Вычислять полином $A(x)$ с границей степени n в заданной точке x_0 можно также путем деления $A(x)$ на полином $(x - x_0)$, в результате чего получатся полином-частное $q(x)$ с границей степени $n - 1$ и остаток r, такой, что

$$A(x) = q(x)(x - x_0) + r .$$

Очевидно, что $A(x_0) = r$. Покажите, как вычислить остаток r и коэффициенты $q(x)$ за время $\Theta(n)$, если заданы точка x_0 и коэффициенты полинома A.

30.1.3

Найдите представление в виде значений в точках полинома $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$, если известно представление в этом виде полинома $A(x) = \sum_{j=0}^{n-1} a_j x^j$; предполагается, что все точки ненулевые.

30.1.4

Докажите, что для однозначного определения полинома с границей степени n необходимо задать n различных пар "точка–значение", т.е. задание меньшего количества различных пар "точка–значение" не позволит определить единственный полином с границей степени n. (*Указание:* используя теорему 30.1, что можно сказать о множестве из $n-1$ пары "точка–значение", к которому добавляется еще одна произвольно выбранная пара "точка–значение"?)

30.1.5

Покажите, как с помощью уравнения (30.5) выполнить интерполяцию за время $\Theta(n^2)$. (*Указание:* сначала следует вычислить представление полинома $\prod_j (x - x_j)$ в форме коэффициентов, а затем разделить его на $(x - x_k)$ для получения числителя каждого члена; см. упр. 30.1.2. Каждый из n знаменателей можно вычислить за время $O(n)$.)

30.1.6

Объясните, что неправильно в "очевидном" подходе к делению представленных в виде значений в точках полиномов, когда значения y одного полинома делятся на соответствующие значения y второго полинома. Рассмотрите отдельно случаи, когда деление полиномов осуществляется без остатка и когда имеется остаток.

30.1.7

Рассмотрим два множества, A и B, каждое из которых содержит n целых чисел в диапазоне от 0 до $10n$. Необходимо вычислить ***декартову сумму*** (Cartesian sum) A и B, определенную следующим образом:

$$C = \{x + y : x \in A \text{ и } y \in B\} .$$

Заметим, что целые числа из множества C заключены в пределах от 0 до $20n$. Требуется найти элементы C и указать, сколько раз каждый элемент C выступает в роли суммы элементов A и B. Покажите, как решить эту задачу за время $O(n \lg n)$. (*Указание:* представьте A и B в виде полиномов с границей степени $10n$.)

30.2. ДПФ и БПФ

В разделе 30.1 утверждалось, что, используя в качестве точек комплексные корни из единицы, можно выполнять вычисление и интерполяцию полиномов за время $\Theta(n \lg n)$. В данном разделе мы дадим определение комплексных корней из единицы и изучим их свойства, определим ДПФ, а затем покажем, как с помощью БПФ можно вычислять ДПФ и обратное ему преобразование за время $\Theta(n \lg n)$.

Комплексные корни из единицы

Комплексным корнем n-й степени из единицы (complex nth root of unity) является комплексное число ω, такое, что

$$\omega^n = 1 \ .$$

Существует ровно n комплексных корней n-й степени из единицы: $e^{2\pi ik/n}$ при $k = 0, 1, \ldots, n-1$. Для интерпретации данной формулы воспользуемся следующим определением экспоненты комплексного числа:

$$e^{iu} = \cos(u) + i\sin(u) \ .$$

На рис. 30.2 показано, что n комплексных корней из единицы равномерно распределены по окружности единичного радиуса с центром в начале координат комплексной плоскости. Значение

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

называется ***главным значением корня n-й степени из единицы*** (the principal nth root of unity); все остальные комплексные корни n-й степени из единицы являются его степенями[2].

Указанные n комплексных корней n-й степени из единицы

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

образуют группу относительно операции умножения (см. раздел 31.3). Эта группа имеет ту же структуру, что и аддитивная группа $(\mathbb{Z}_n, +)$ по модулю n, поскольку из $\omega_n^n = \omega_n^0 = 1$ следует, что $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Аналогично $\omega_n^{-1} = \omega_n^{n-1}$. Основные свойства комплексных корней n-й степени из единицы приведены в следующих леммах.

---

[2]Многие авторы определяют ω_n иначе: $\omega_n = e^{-2\pi i/n}$. Это альтернативное определение обычно используется в обработке сигналов. Лежащие в основе математические концепции в основном одинаковы для обоих определений.

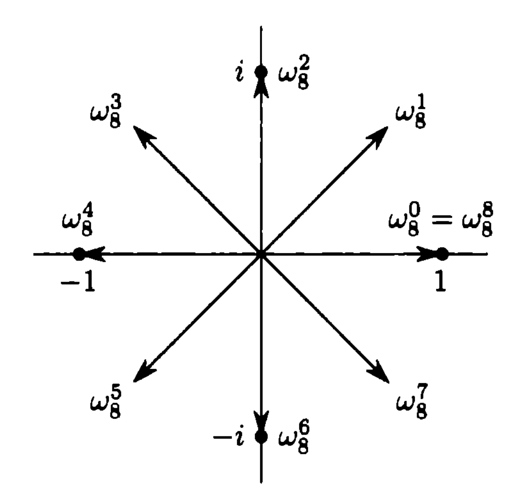


Рис. 30.2. Значения $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ на комплексной плоскости, где $\omega_8 = e^{2\pi i/8}$ — главное значение корня восьмой степени из единицы.

Лемма 30.3 (Лемма о сокращении)

Для любых целых чисел $n \geq 0$, $k \geq 0$ и $d > 0$

$$\omega_{dn}^{dk} = \omega_n^k . \tag{30.7}$$

Доказательство. Данная лемма непосредственно следует из уравнения (30.6), поскольку

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k . \end{aligned}$$ ■

Следствие 30.4

Для любого четного целого $n > 0$

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Доказательство. Доказательство предлагается провести самостоятельно в качестве упр. 30.2.1. ■

Лемма 30.5 (Лемма о делении пополам)

Если $n > 0$ четное, то квадраты n комплексных корней n-й степени из единицы представляют собой $n/2$ корней $n/2$-й степени из единицы.

Доказательство. Согласно лемме о сокращении для любого неотрицательного целого k справедливо $(\omega_n^k)^2 = \omega_{n/2}^k$. Заметим, что если возвести в квадрат все комплексные корни n-й степени из единицы, то каждый корень $n/2$-й степени из

единицы будет получен в точности дважды, поскольку

$$\begin{aligned}(\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 \ .\end{aligned}$$

Таким образом, квадраты ω_n^k и $\omega_n^{k+n/2}$ одинаковы. Это свойство можно также доказать с помощью следствия 30.4: из $\omega_n^{n/2} = -1$ вытекает $\omega_n^{k+n/2} = -\omega_n^k$, следовательно, $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. ■

Как мы увидим далее, лемма о делении пополам играет исключительно важную роль в применении декомпозиции для преобразования полиномов из представления в форме коэффициентов в форму значений в точках и обратно, поскольку она гарантирует, что рекурсивные подзадачи имеют половинный размер.

Лемма 30.6 (Лемма о суммировании)

Для любого целого $n \geq 1$ и ненулевого целого k, не кратного n,

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0 \ .$$

Доказательство. Уравнение (A.5) применимо как к комплексным, так и к действительным числам, так что имеем

$$\begin{aligned}\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0 \ .\end{aligned}$$

Поскольку мы требуем, чтобы k не было кратно n, и поскольку $\omega_n^k = 1$ только тогда, когда k делится на n, это гарантирует, что знаменатель не равен нулю. ■

Дискретное преобразование Фурье

Вспомним, что мы хотим вычислить полином

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

с границей степени n в точках $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (которые представляют собой n комплексных корней n-й степени из единицы)[3]. Предположим, что полином A задан в форме коэффициентов: $a = (a_0, a_1, \ldots, a_{n-1})$. Определим результаты y_k, где $k = 0, 1, \ldots, n-1$, с помощью формулы

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} . \end{aligned} \tag{30.8}$$

Вектор $y = (y_0, y_1, \ldots, y_{n-1})$ представляет собой ***дискретное преобразование Фурье***, ДПФ (discrete Fourier transform — DFT) вектора коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$. Можно также записать $y = \text{ДПФ}_n(a)$.

Быстрое преобразование Фурье

С помощью метода, известного как ***быстрое преобразование Фурье***, БПФ (Fast Fourier Transform — FFT), основанного на использовании специальных свойств комплексных корней из единицы, $\text{ДПФ}_n(a)$ можно вычислять за время $\Theta(n \lg n)$, в отличие от метода непосредственного преобразования, имеющего время работы $\Theta(n^2)$. Будем считать, что n является точной степенью 2. Хотя известны методы, работающие и с другими значениями, в нашей книге они не рассматриваются.

В методе БПФ применяется стратегия декомпозиции, в которой отдельно используются коэффициенты полинома $A(x)$ с четными и нечетными индексами, чтобы определить два новых полинома, $A^{[0]}(x)$ и $A^{[1]}(x)$, с границей степени $n/2$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

Заметим, что $A^{[0]}$ содержит все коэффициенты $A(x)$ с четными индексами (двоичное представление этих индексов заканчивается цифрой 0), а $A^{[1]}$ содержит все коэффициенты с нечетными индексами (двоичное представление которых закан-

---

[3] Здесь n на самом деле представляет собой величину, которая в разделе 30.1 обозначалась как $2n$, поскольку, прежде чем выполнять вычисление, границы степеней заданных полиномов были удвоены. Таким образом, при умножении полиномов речь в действительности идет о комплексных корнях из единицы $2n$-й степени.

чивается цифрой 1). Отсюда следует, что

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \ , \tag{30.9}$$

так что задача вычисления $A(x)$ в точках $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ сводится к следующим задачам:

1. вычисление полиномов $A^{[0]}(x)$ и $A^{[1]}(x)$ с границей степени $n/2$ в точках

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 \ ; \tag{30.10}$$

2. а затем объединение полученных результатов в соответствии с уравнением (30.9).

Согласно лемме о делении пополам список значений (30.10) содержит не n различных значений, а только $n/2$ комплексных корней степени $n/2$ из единицы, причем каждый корень встречается в списке ровно дважды. Следовательно, полиномы $A^{[0]}$ и $A^{[1]}$ с границей степени $n/2$ рекурсивно вычисляются в $n/2$ комплексных корнях $n/2$-й степени из единицы. Эти подзадачи имеют точно такой же вид, как и исходная задача, но их размерность вдвое меньше. Таким образом, мы успешно свели вычисление n-элементного ДПФ$_n$ к вычислению двух $n/2$-элементных ДПФ$_{n/2}$. Такая декомпозиция является основой следующего рекурсивного алгоритма БПФ, который вычисляет ДПФ n-элементного вектора $a = (a_0, a_1, \ldots, a_{n-1})$, где n представляет собой степень 2.

```
RECURSIVE-FFT(a)
1  n = a.length                    // n является степенью 2
2  if n == 1
3      return a
4  ω_n = e^{2πi/n}
5  ω = 1
6  a^[0] = (a_0, a_2, ..., a_{n-2})
7  a^[1] = (a_1, a_3, ..., a_{n-1})
8  y^[0] = RECURSIVE-FFT(a^[0])
9  y^[1] = RECURSIVE-FFT(a^[1])
10 for k = 0 to n/2 - 1
11     y_k = y_k^[0] + ω y_k^[1]
12     y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13     ω = ω ω_n
14 return y                        // Считаем, что y — вектор-столбец
```

Процедура RECURSIVE-FFT работает следующим образом. Строки 2 и 3 представляют базис данной рекурсии: ДПФ одного элемента является самим этим

элементом, следовательно, в этом случае

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \; . \end{aligned}$$

В строках 6 и 7 определяется вектор коэффициентов для полиномов $A^{[0]}$ и $A^{[1]}$. Строки 4, 5 и 13 гарантируют, что ω обновляется надлежащим образом, т.е. всякий раз, когда выполняются строки 11 и 12, мы имеем $\omega = \omega_n^k$. (Сохранение текущего значения ω от одной итерации к другой позволяет экономить время по сравнению с многократным вычислением ω_n^k с нуля с помощью цикла **for**.) В строках 8 и 9 выполняются рекурсивные вычисления ДПФ$_{n/2}$ с установкой для $k = 0, 1, \ldots, n/2 - 1$ следующих значений

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) \; , \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) \; , \end{aligned}$$

или, поскольку согласно лемме о сокращении $\omega_{n/2}^k = \omega_n^{2k}$,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) \; , \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) \; . \end{aligned}$$

В строках 11 и 12 выполняется комбинация результатов рекурсивных вычислений ДПФ$_{n/2}$. Для $y_0, y_1, \ldots, y_{n/2-1}$ строка 11 дает

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) && \text{(согласно (30.9))} \; . \end{aligned}$$

Для $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, полагая $k = 0, 1, \ldots, n/2 - 1$, в строке 12 получаем

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && \text{(так как } \omega_n^{k+(n/2)} = -\omega_n^k\text{)} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && \text{(так как } \omega_n^{2k+n} = \omega_n^{2k}\text{)} \\ &= A(\omega_n^{k+(n/2)}) && \text{(согласно (30.9))} \; . \end{aligned}$$

Таким образом, возвращаемый процедурой RECURSIVE-FFT вектор y действительно представляет собой ДПФ исходного вектора a.

В строках 11 и 12 выполняется умножение каждого значения $y_k^{[1]}$ на ω_n^k для $k = 0, 1, \ldots, n/2 - 1$. Строка 11 прибавляет это произведение к $y_k^{[0]}$, а строка 12

вычитает его. Поскольку мы используем каждый множитель ω_n^k как в положительном, так и в отрицательном виде, множители ω_n^k называются ***поворачивающими множителями*** (twiddle factors).

Чтобы определить время работы процедуры RECURSIVE-FFT, заметим, что, за исключением рекурсивных вызовов, каждый вызов занимает время $\Theta(n)$, где n — длина исходного вектора. Таким образом, рекуррентное соотношение для времени выполнения имеет вид

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Итак, используя быстрое преобразование Фурье, можно вычислить полином с границей степени n в комплексных корнях единицы n-й степени за время $\Theta(n \lg n)$.

Интерполяция в комплексных корнях единицы

Завершим рассмотрение схемы умножения полиномов, показав, как выполняется интерполяция комплексных корней из единицы некоторым полиномом, что позволяет перейти от формы значений в точках обратно к форме коэффициентов. Для этого ДПФ записывается в виде матричного уравнения, после чего выполняется поиск обратной матрицы.

Из уравнения (30.4) можно записать ДПФ как произведение матриц $y = V_n a$, где V_n является матрицей Вандермонда, содержащей соответствующие степени ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

(k, j)-м элементом матрицы V_n является ω_n^{kj} при $j, k = 0, 1, \ldots, n-1$. Показатели степеней элементов матрицы V_n образуют таблицу умножения.

Для выполнения обратной операции, которую запишем как $a = \text{ДПФ}_n^{-1}(y)$, нужно умножить матрицу V_n^{-1}, обратную к V_n, на y.

Теорема 30.7

(j, k)-й элемент матрицы V_n^{-1} представляет собой ω_n^{-kj}/n для $j, k = 0, 1, \ldots, n-1$.

Доказательство. Покажем, что $V_n^{-1}V_n = I_n$, единичной матрице размера $n \times n$. Рассмотрим (j, j')-й элемент $V_n^{-1}V_n$:

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1}(\omega_n^{-kj}/n)(\omega_n^{kj'}) = \sum_{k=0}^{n-1}\omega_n^{k(j'-j)}/n .$$

Эта сумма равна 1 при $j' = j$ и 0 — в противном случае согласно лемме о суммировании (лемма 30.6). Чтобы применить эту лемму, заметим, что $-(n-1) \leq j' - j \leq n - 1$, так что $j' - j$ не делится на n. ■

Для заданной обратной матрицы V_n^{-1} обратное преобразование ДПФ$_n^{-1}(y)$ вычисляется по формуле

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k\omega_n^{-kj} \qquad (30.11)$$

для $j = 0, 1, \ldots, n-1$. Сравнивая уравнения (30.8) и (30.11), мы видим, что если модифицировать алгоритм БПФ путем обмена ролей a и y, замены ω_n на ω_n^{-1} и деления каждого элемента результата на n, то получится обратное ДПФ (см. упр. 30.2.4). Следовательно, ДПФ$_n^{-1}$ также можно вычислить за время $\Theta(n \lg n)$.

Таким образом, с помощью прямого и обратного БПФ можно преобразовывать полином с границей степени n из формы коэффициентов в форму значений в точках и обратно за время $\Theta(n \lg n)$. Применительно к умножению полиномов мы показали следующее.

Теорема 30.8 (Теорема о свертке)

Для любых двух векторов a и b длиной n, где n является степенью 2, справедливо соотношение

$$a \otimes b = \text{ДПФ}_{2n}^{-1}(\text{ДПФ}_{2n}(a) \cdot \text{ДПФ}_{2n}(b)) ,$$

где векторы a и b дополняются нулями до длины $2n$, а "$\cdot$" обозначает покомпонентное произведение двух $2n$-элементных векторов. ■

Упражнения

30.2.1

Докажите следствие 30.4.

30.2.2

Вычислите ДПФ вектора $(0, 1, 2, 3)$.

30.2.3

Выполните упр. 30.1.1, используя схему со временем выполнения $\Theta(n \lg n)$.

30.2.4

Напишите псевдокод для вычисления ДПФ$_n^{-1}$ за время $\Theta(n \lg n)$.

30.2.5

Опишите обобщение процедуры БПФ для случая, когда n является степенью 3. Приведите рекуррентное соотношение для времени выполнения и решите его.

30.2.6 ★

Предположим, что вместо выполнения n-элементного БПФ над полем комплексных чисел (где n четно) мы используем кольцо $\mathbb{Z}_m$ целых чисел по модулю $m = 2^{tn/2+1}$, где t — произвольное положительное целое число. Используйте $\omega = 2^t$ вместо ω_n в качестве главного значения корня n-й степени из единицы по модулю m. Докажите, что прямое и обратное ДПФ в этой системе является вполне определенным.

30.2.7

Покажите, как для заданного списка значений $z_0, z_1, \ldots, z_{n-1}$ (возможно, с повторениями) найти коэффициенты полинома $P(x)$ с границей степени $n+1$, который имеет нули только в точках $z_0, z_1, \ldots, z_{n-1}$ (возможно, с повторениями). Процедура должна выполняться за время $O(n \lg^2 n)$. (*Указание:* полином $P(x)$ имеет нулевое значение в точке z_j тогда и только тогда, когда $P(x)$ кратен $(x - z_j)$.)

30.2.8 ★

Чирп-преобразованием (chirp transform) некоторого вектора $a = (a_0, a_1, \ldots, a_{n-1})$ является вектор $y = (y_0, y_1, \ldots, y_{n-1})$, где $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$, а z — произвольное комплексное число. Таким образом, ДПФ является частным случаем чирп-преобразования при $z = \omega_n$. Покажите, как за время $O(n \lg n)$ вычислить чирп-преобразование для любого комплексного числа z. (*Указание:* воспользуйтесь уравнением

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right) ,$$

чтобы рассматривать чирп-преобразование как свертку.)

30.3. Эффективные реализации БПФ

Поскольку практические приложения ДПФ, такие как обработка сигналов, требуют максимальной скорости, в данном разделе мы рассмотрим две наиболее эффективные реализации БПФ. Сначала будет описана итеративная версия алгоритма БПФ, время выполнения которой составляет $\Theta(n \lg n)$, однако при этом

в Θ скрыта меньшая константа, чем в случае рекурсивной реализации, предложенной в разделе 30.2. После этого мы вернемся к интуитивным соображениям, приведшим нас к итеративной реализации, чтобы разработать эффективную параллельную схему БПФ.

Итеративная реализация БПФ

Прежде всего заметим, что цикл **for** в строках 10–13 процедуры RECURSIVE-FFT дважды вычисляет значение $\omega_n^k y_k^{[1]}$. В терминологии компиляторов такое значение называется ***общим подвыражением*** (common subexpression). Можно изменить цикл так, чтобы вычислять это значение только один раз, сохраняя его во временной переменной t.

for $k = 0$ **to** $n/2 - 1$
    $t = \omega\, y_k^{[1]}$
    $y_k = y_k^{[0]} + t$
    $y_{k+(n/2)} = y_k^{[0]} - t$
    $\omega = \omega\, \omega_n$

Выполняемая в данном цикле операция (умножение поворачивающего множителя $\omega = \omega_n^k$ на $y_k^{[1]}$, сохранение произведения в переменной t, прибавление и вычитание t из $y_k^{[0]}$) известна как ***преобразование бабочки*** (butterfly operation), схематически представленное на рис. 30.3.

Теперь покажем, как сделать алгоритм БПФ итеративным, а не рекурсивным. На рис. 30.4 мы организовали входные векторы рекурсивных вызовов в обращении к процедуре RECURSIVE-FFT в древовидную структуру, в которой первый вызов выполняется для $n = 8$. Данное дерево содержит по одному узлу для каждого вызова процедуры, помеченному соответствующим входным вектором. Каждое обращение к процедуре RECURSIVE-FFT производит два рекурсивных вызова, пока не получит 1-элементный вектор. Первый вызов находится в левом дочернем узле, а второй — в правом.

Рассматривая дерево, можно заметить, что если бы можно было расположить элементы исходного вектора a в том порядке, в котором они появляются в листьях дерева, то можно было бы увидеть выполнение процедуры RECURSIVE-FFT

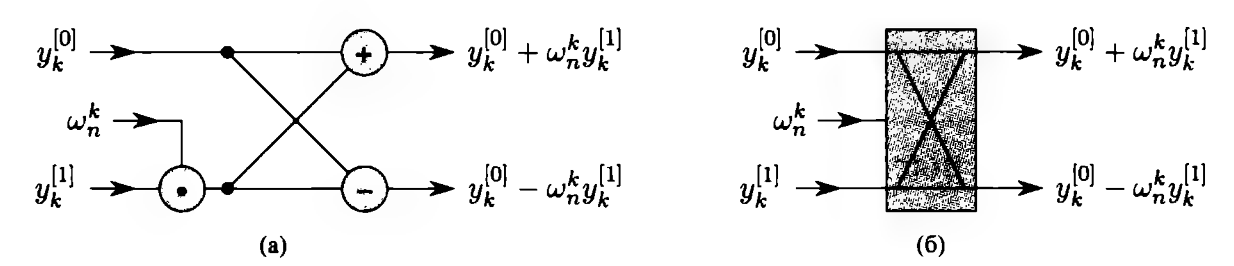


Рис. 30.3. Преобразование бабочки. **(а)** Слева поступают два входных значения, поворачивающий множитель ω_n^k умножается на $y_k^{[1]}$ и соответствующие сумма и разность выводятся справа. **(б)** Упрощенная схема преобразования бабочки. Это представление будет использоваться в параллельной схеме БПФ.

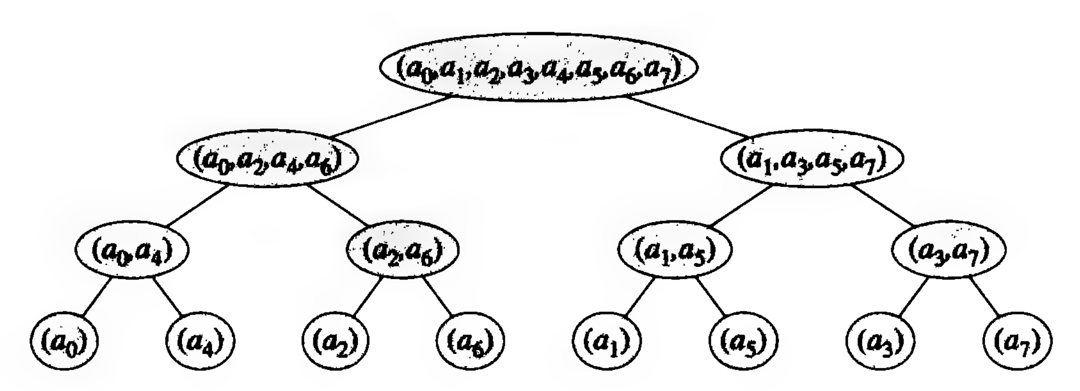


Рис. 30.4. Дерево входных векторов рекурсивных вызовов процедуры RECURSIVE-FFT. Начальный вызов выполняется для $n = 8$.

в восходящем направлении, а не в нисходящем. Сначала берутся пары элементов, с помощью одного преобразования бабочки вычисляется ДПФ каждой пары, и пара заменяется вычисленным ДПФ. В результате получается вектор, содержащий $n/2$ 2-элементных ДПФ. Затем эти $n/2$ ДПФ объединяются в пары, и с помощью двух преобразований бабочки вычисляются ДПФ для каждых четырех элементов вектора, объединенных в четверки, после чего два 2-элементных ДПФ заменяются одним 4-элементным ДПФ. Теперь вектор содержит $n/4$ 4-элементных ДПФ. Процесс продолжается до тех пор, пока не получится два $n/2$-элементных ДПФ, которые с помощью $n/2$ преобразований бабочки можно объединить в конечное n-элементное ДПФ.

Чтобы превратить этот восходящий подход в код, используем массив $A[0\,..\,n-1]$, который изначально хранит элементы входного вектора a в том порядке, в котором они появляются в листьях дерева на рис. 30.4. (Позже мы покажем, как определить этот порядок, который называется поразрядно обратной перестановкой (bit-reversal permutation).) Поскольку объединение в пары необходимо выполнять на каждом уровне дерева, введем в качестве счетчика уровней переменную s, которая изменяется от 1 (на нижнем уровне, где составляются пары для вычисления 2-элементных ДПФ) до $\lg n$ (на вершине, где объединяются два $n/2$-элементных ДПФ для получения окончательного результата). Таким образом, алгоритм имеет следующую структуру.

1 **for** $s = 1$ **to** $\lg n$
2   **for** $k = 0$ **to** $n - 1$ **by** 2^s
3   объединить два 2^{s-1}-элементных ДПФ в
            $A[k\,..\,k + 2^{s-1} - 1]$ и $A[k + 2^{s-1}\,..\,k + 2^s - 1]$
            в одно 2^s-элементное ДПФ в $A[k\,..\,k + 2^s - 1]$

Тело цикла (строка 3) можно описать более подробным и точным псевдокодом. Скопируем цикл **for** из процедуры RECURSIVE-FFT, отождествив $y^{[0]}$ с $A[k\,..\,k + 2^{s-1} - 1]$ и $y^{[1]}$ с $A[k + 2^{s-1}\,..\,k + 2^s - 1]$. Поворачивающий множитель, используемый в каждом преобразовании бабочки, зависит от значения s; он представляет собой степень ω_m, где $m = 2^s$. (Мы ввели переменную m исключительно для того, чтобы формула была удобочитаемой.) Введем еще одну временную переменную u, которая позволит выполнять преобразование бабочки на месте, без

привлечения дополнительной памяти. Заменив строку 3 в общей схеме телом цикла, получим следующий псевдокод, образующий базис параллельной реализации, которая будет представлена позже. В данном коде сначала вызывается вспомогательная процедура BIT-REVERSE-COPY(a, A), копирующая вектор a в массив A в том исходном порядке, в котором нам нужны эти значения.

```
ITERATIVE-FFT(a)
BIT-REVERSE-COPY(a, A)
n = a.length                 // n является степенью 2
for s = 1 to lg n
    m = 2^s
    ω_m = e^{2πi/m}
    for k = 0 to n − 1 by m
        ω = 1
        for j = 0 to m/2 − 1
            t = ω A[k + j + m/2]
            u = A[k + j]
            A[k + j] = u + t
            A[k + j + m/2] = u − t
            ω = ω ω_m
return A
```

Каким образом процедура BIT-REVERSE-COPY размещает элементы входного вектора a в массиве A в требуемом порядке? Порядок следования листьев на рис. 30.4 представляет собой ***обратную перестановку***, или ***реверс битов*** (bit-reversal permutation), т.е. если $\text{rev}(k)$ — $\lg n$-битовое целое число, образованное путем размещения битов исходного двоичного представления k в обратном порядке ("задом наперед"), то элемент a_k следует поместить в массиве на место $A[\text{rev}(k)]$. На рис. 30.4, например, листья находятся в следующем порядке: $0, 4, 2, 6, 1, 5, 3, 7$; в двоичном представлении эта последовательность записывается как $000, 100, 010, 110, 001, 101, 011, 111$. Если же записать биты каждого значения в обратном порядке, получится обычная числовая последовательность: $000, 001, 010, 011, 100, 101, 110, 111$. Чтобы убедиться в том, что нам нужна именно обратная перестановка битов, заметим, что на верхнем уровне дерева индексы, младший бит которых равен 0, помещаются в левое поддерево, а индексы, младший бит которых равен 1, помещаются в правое поддерево. Исключая на каждом уровне младший бит, мы продолжаем процесс вниз по дереву, пока не получим в листовых узлах порядок, заданный обратной перестановкой битов.

Поскольку функция $\text{rev}(k)$ вычисляется очень легко, процедуру BIT-REVERSE-COPY можно записать очень просто.

```
BIT-REVERSE-COPY(a, A)
n = a.length
for k = 0 to n − 1
    A[rev(k)] = a_k
```

Итеративная реализация БПФ выполняется за время $\Theta(n \lg n)$. Вызов процедуры BIT-REVERSE-COPY(a, A) выполняется за время $O(n \lg n)$, поскольку проводится n итераций, а целое число от 0 до $n - 1$, содержащее $\lg n$ бит, можно привести к обратному порядку за время $O(\lg n)$[4]. (На практике мы обычно заранее знаем начальное значение n, поэтому можно составить таблицу, отображающую k в $\text{rev}(k)$, в результате процедура BIT-REVERSE-COPY будет выполняться за время $\Theta(n)$ с малой скрытой константой. В качестве альтернативного подхода можно использовать схему амортизированного обратного бинарного битового счетчика, описанную в задаче 17.1.) Чтобы завершить доказательство того факта, что процедура ITERATIVE-FFT выполняется за время $\Theta(n \lg n)$, покажем, что число $L(n)$ выполнений тела внутреннего цикла (строки 8–13) равно $\Theta(n \lg n)$. Цикл **for** (строки 6–13) повторяется $n/m = n/2^s$ раз для каждого значения s, а внутренний цикл (строки 8–13) повторяется $m/2 = 2^{s-1}$ раз. Отсюда

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n) \; . \end{aligned}$$

Параллельная схема БПФ

Чтобы получить эффективный параллельный алгоритм БПФ, можно воспользоваться многими из свойств, которые позволили нам реализовать эффективный итеративный алгоритм БПФ. Мы будем представлять параллельный алгоритм БПФ в виде схемы. На рис. 30.5 показана параллельная схема, которая вычисляет БПФ для n входных значений при $n = 8$. Она начинается с поразрядной обратной перестановки исходных значений, затем следуют $\lg n$ этапов, на каждом из которых параллельно выполняется $n/2$ операций бабочки. Таким образом, ***глубина*** (depth) схемы — максимальное количество вычислительных элементов между любыми входами и выходами — составляет $\Theta(\lg n)$.

Крайняя слева часть схемы PARALLEL-FFT выполняет поразрядно обратную перестановку, а остальная часть имитирует итеративную процедуру ITERATIVE-FFT. Поскольку каждая итерация внешнего цикла **for** выполняет $n/2$ независимых преобразований бабочки, в данной схеме они выполняются параллельно. Значение s в каждой итерации ITERATIVE-FFT соответствует каскаду преобразований бабочки, показанному на рис. 30.5. В пределах этапа $s = 1, 2, \ldots, \lg n$ имеется $n/2^s$ групп преобразований бабочки (соответствующих различным значениям k процедуры ITERATIVE-FFT), в каждой группе выполняется 2^{s-1} операций (соответствующих различным значениям j процедуры ITERATIVE-FFT).

---

[4]Интересные реализации реверса битов можно найти в разделе 7.1 книги Г. Уоррен. *Алгоритмические трюки для программистов*. — М.: И.Д. "Вильямс", 2003. — *Примеч. ред.*

31 Зак. 3726

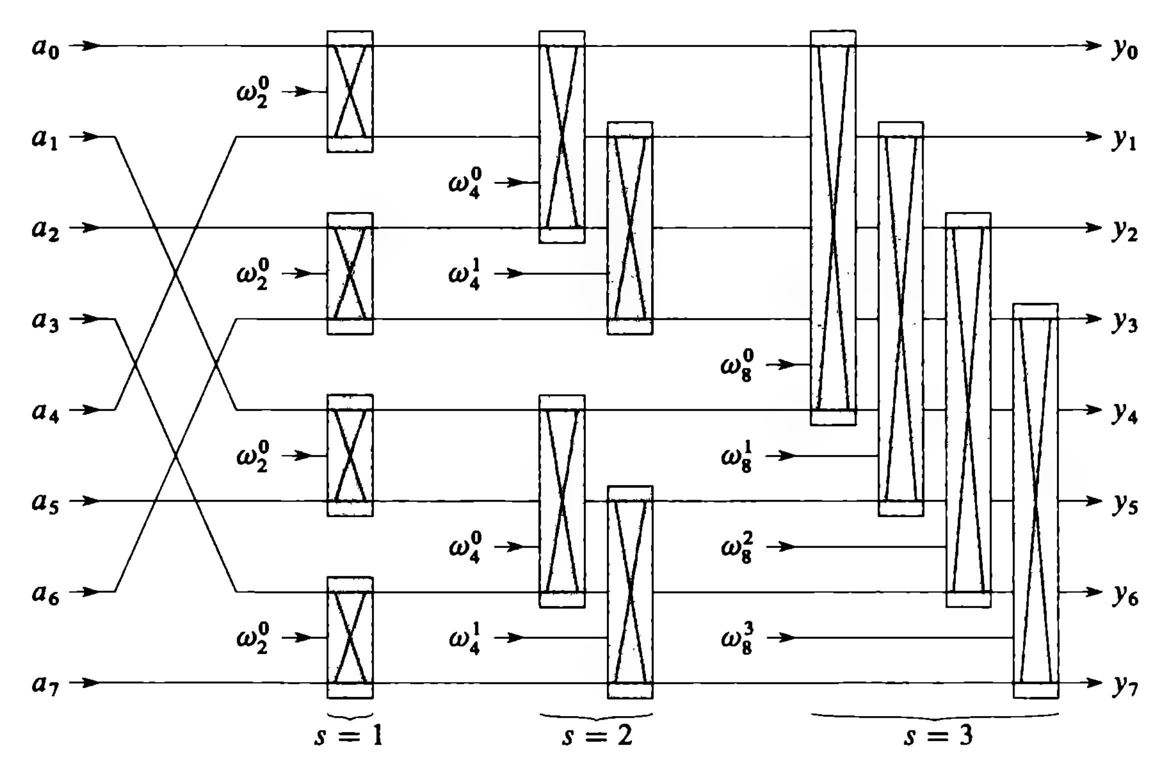


Рис. 30.5. Схема параллельного вычисления БПФ (в данном случае — для $n = 8$ входов). Каждое преобразование бабочки получает в качестве входных данных значения, поступающие по двум проводам, вместе с поворачивающим множителем и передает полученные значения на два выходящих провода. Различные этапы каскада преобразований бабочки нумеруются в соответствии с итерациями самого внешнего цикла процедуры ITERATIVE-FFT. При проходе через бабочку в вычислениях участвуют только нижний и верхний провода; провода, проходящие через середину бабочки, никак не взаимодействуют с ней, и их значения также не меняются данной бабочкой. Например, верхняя бабочка для $s = 2$ ничего не делает с проводом 1 (которому соответствует выходная переменная, обозначенная как y_1); ее вход и выход находятся в проводах 0 и 2 (обозначенных соответственно как y_0 и y_2). Для вычисления БПФ для n переменных ввода требуется схема глубиной $\Theta(\lg n)$, всего содержащая $\Theta(n \lg n)$ операций бабочки.

Преобразования бабочки, показанные на рис. 30.5, соответствуют операциям во внутреннем цикле (строки 9–12 процедуры ITERATIVE-FFT). Заметим также, что используемые в бабочках поворачивающие множители соответствуют поворачивающим множителям, используемым в процедуре ITERATIVE-FFT: на этапе s используются значения $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, где $m = 2^s$.

Упражнения

30.3.1

Покажите, как процедура ITERATIVE-FFT вычисляет ДПФ входного вектора $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3.2

Покажите, как реализовать алгоритм БПФ, в котором обратная перестановка битов выполняется в конце, а не в начале процесса вычислений. (*Указание:* рассмотрите обратное ДПФ.)

30.3.3

Сколько раз процедура ITERATIVE-FFT вычисляет поворачивающие множители на каждом этапе? Перепишите процедуру ITERATIVE-FFT так, чтобы поворачивающие множители на этапе s вычислялись только 2^{s-1} раз.

30.3.4 ★

Предположим, что сумматоры в преобразованиях бабочки в схеме БПФ иногда дают сбои, приводящие к нулевому результату независимо от подаваемых на вход значений. Предположим, что сбой произошел ровно в одном сумматоре, однако неизвестно, в каком именно. Опишите, как можно быстро выявить неисправный сумматор путем тестирования всей БПФ-схемы с помощью различных входных данных и изучения выходных. Насколько эффективен ваш метод?

Задачи

30.1. Умножение посредством декомпозиции

а. Покажите, как умножить два линейных полинома, $ax + b$ и $cx + d$, используя только три операции умножения. (*Указание:* одно из умножений — $(a + b) \cdot (c + d)$.)

б. Приведите два алгоритма декомпозиции для умножения полиномов с границей степени n, время выполнения которых — $\Theta(n^{\lg 3})$. В первом алгоритме следует разделить коэффициенты исходного полинома на старшие и младшие, а во втором — на четные и нечетные.

в. Покажите, как перемножить два n-битовых целых числа за $O(n^{\lg 3})$ шагов, где каждый шаг работает с не превышающим некоторую константу количеством однобитовых значений.

30.2. Матрицы Теплица

Матрицей Теплица (Toeplitz matrix) называется матрица $A = (a_{ij})$ размером $n \times n$, в которой $a_{ij} = a_{i-1,j-1}$ для $i = 2, 3, \ldots, n$ и $j = 2, 3, \ldots, n$.

а. Всегда ли сумма двух матриц Теплица является матрицей Теплица? Что можно сказать об их произведении?

б. Каким должно быть представление матриц Теплица, чтобы сложение двух матриц Теплица размером $n \times n$ можно было выполнить за время $O(n)$?

в. Предложите алгоритм умножения матрицы Теплица размером $n \times n$ на вектор длиной n со временем работы $O(n \lg n)$. Воспользуйтесь представлением, полученным при решении предыдущего пункта данной задачи.

г. Предложите эффективный алгоритм умножения двух матриц Теплица размером $n \times n$ и проанализируйте время его выполнения.

30.3. Многомерное быстрое преобразование Фурье

Можно обобщить одномерное дискретное преобразование Фурье, определенное уравнением (30.8), на d-мерный случай. Пусть на вход подается d-мерный массив $A = (a_{j_1,j_2,\ldots,j_d})$ с размерностями $n_1, n_2, \ldots, n_d$, которые удовлетворяют соотношению $n_1 n_2 \cdots n_d = n$. Определим d-мерное ДПФ следующим образом:

$$y_{k_1,k_2,\ldots,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,\ldots,j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

для $0 \le k_1 < n_1$, $0 \le k_2 < n_2, \ldots, 0 \le k_d < n_d$.

а. Покажите, что можно вычислить d-мерное ДПФ путем поочередного вычисления одномерных ДПФ. Таким образом, сначала вычисляется n/n_1 отдельных одномерных ДПФ вдоль первого измерения. Затем, используя результат ДПФ вдоль первого измерения в качестве входных данных, вычисляется n/n_2 одномерных ДПФ вдоль второго измерения. Используя этот результат в качестве входных данных, вычисляется n/n_3 отдельных ДПФ вдоль третьего измерения, и так до последнего измерения d.

б. Покажите, что порядок следования измерений не имеет значения, т.е. что можно вычислять d-мерное ДПФ путем вычисления одномерных ДПФ для каждого из d измерений в произвольном порядке.

в. Покажите, что если каждое одномерное ДПФ вычислять с помощью быстрого преобразования Фурье, то суммарное время вычисления d-мерного ДПФ составляет $O(n \lg n)$ независимо от d.

30.4. Вычисление всех производных полинома в точке

Пусть задан полином $A(x)$ с границей степени n; t-я производная этого полинома определяется формулой

$$A^{(t)}(x) = \begin{cases} A(x) \ , & \text{если } t = 0 \ , \\ \frac{d}{dx} A^{(t-1)}(x) \ , & \text{если } 1 \le t \le n-1 \ , \\ 0 \ , & \text{если } t \ge n \ . \end{cases}$$

Пусть заданы представление в форме коэффициентов $(a_0, a_1, \ldots, a_{n-1})$ полинома $A(x)$ и некоторая точка x_0. Требуется найти $A^{(t)}(x_0)$ для $t = 0, 1, \ldots, n-1$.

а. Пусть заданы коэффициенты $b_0, b_1, \ldots, b_{n-1}$, такие, что

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j \ .$$

Покажите, как вычислить $A^{(t)}(x_0)$ для $t = 0, 1, \ldots, n-1$ за время $O(n)$.

б. Поясните, как найти $b_0, b_1, \ldots, b_{n-1}$ за время $O(n \lg n)$ при заданных значениях $A(x_0 + \omega_n^k)$ для $k = 0, 1, \ldots, n-1$.

в. Докажите, что

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right) ,$$

где $f(j) = a_j \cdot j!$, а

$$g(l) = \begin{cases} x_0^{-l}/(-l)! , & \text{если } -(n-1) \le l \le 0 , \\ 0 , & \text{если } 1 \le l \le n-1 . \end{cases}$$

г. Поясните, как вычислить $A(x_0 + \omega_n^k)$ для $k = 0, 1, \ldots, n-1$ за время $O(n \lg n)$. Сделайте вывод, что все нетривиальные производные $A(x)$ в точке x_0 можно вычислить за время $O(n \lg n)$.

30.5. Вычисление полинома в нескольких точках

Как уже говорилось, задачу вычисления полинома с границей степени n в единственной точке можно решить с помощью схемы Горнера за время $O(n)$. Мы также показали, как с помощью БПФ такой полином можно вычислить во всех n комплексных корнях единицы за время $O(n \lg n)$. Теперь покажем, как вычислить полином с границей степени n в произвольных n точках за время $O(n \lg^2 n)$.

Для этого будем считать, что остаток при делении одного такого полинома на другой можно вычислить за время $O(n \lg n)$ (этот результат мы принимаем без доказательства). Например, остаток при делении $3x^3 + x^2 - 3x + 1$ на $x^2 + x + 2$ равен

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5 .$$

Пусть заданы представление полинома $A(x) = \sum_{k=0}^{n-1} a_k x^k$ в форме коэффициентов и n точек $x_0, x_1, \ldots, x_{n-1}$ и пусть необходимо вычислить n значений $A(x_0), A(x_1), \ldots, A(x_{n-1})$. Для $0 \le i \le j \le n-1$ определим полиномы $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ и $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Заметим, что полином $Q_{ij}(x)$ имеет степень не выше $j - i$.

а. Докажите, что $A(x) \bmod (x - z) = A(z)$ для любой точки z.

б. Докажите, что $Q_{kk}(x) = A(x_k)$ и что $Q_{0,n-1}(x) = A(x)$.

в. Докажите, что для $i \le k \le j$ справедливы соотношения $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ и $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

г. Предложите алгоритм вычисления $A(x_0), A(x_1), \ldots, A(x_{n-1})$ за время $O(n \lg^2 n)$.

30.6. БПФ с использованием модульной арифметики

Согласно определению при вычислении дискретного преобразования Фурье требуется использовать комплексные числа, что может привести к потере точности из-за ошибок округления. Для некоторых задач известно, что ответы содержат только целые значения, поэтому желательно использовать вариант БПФ, основанный на модульной арифметике, чтобы гарантировать, что ответ вычисляется точно. Примером такой задачи является умножение двух полиномов с целыми коэффициентами. В упр. 30.2.6 предлагался подход, в котором используются модули длиной $\Omega(n)$ бит для вычисления ДПФ по n точкам. В данной задаче предлагается иной подход, в котором используются модули более подходящей длины $O(\lg n)$; для его понимания следует ознакомиться с материалом главы 31. Предполагается, что n является степенью 2.

а. Предположим, что требуется найти наименьшее значение k, такое, что $p = kn + 1$ является простым числом. Предложите простое эвристическое доказательство того, что k примерно равно $\lg n$. (Значение k может быть больше или меньше, но в среднем придется рассмотреть $O(\lg n)$ возможных значений k.) Как ожидаемая длина p соотносится с длиной n?

Предположим, что g является генератором $\mathbb{Z}_p^*$, и пусть $w = g^k \bmod p$.

б. Докажите, что ДПФ и обратное ДПФ являются вполне определенными обратными операциями по модулю p, если в качестве w используется главное значение корня n-й степени из единицы.

в. Покажите, как БПФ и обратное ему преобразование по модулю p могут выполняться за время $O(n \lg n)$, если при этом операции со словами длиной $O(\lg n)$ бит выполняются за единичное время. (Значения p и w считаются заданными.)

г. Вычислите ДПФ по модулю $p = 17$ для вектора $(0, 5, 3, 7, 7, 2, 1, 6)$. Учтите, что генератором $\mathbb{Z}_{17}^*$ является $g = 3$.

Заключительные замечания

Подробное рассмотрение БПФ можно найти в книге Ван Лона (Van Loan) [341]. В работах Пресса (Press), Тукольски (Teukolsky), Веттерлинга (Vetterling) и Фланнери (Flannery) [281,282] имеется хорошее описание БПФ и его приложений. Прекрасное введение в обработку сигналов — область широкого применения БПФ — предлагается в работах Оппенгейма (Oppenheim) и Шафера (Schafer) [264] и Оппенгейма и Уиллски (Willsky) [265]. В книге Оппенгейма и Шафера также описана работа в ситуации, когда n не является целой степенью 2.

Анализ Фурье не ограничивается одномерными данными. Он широко используется в обработке изображений для анализа данных в двух и более измерениях.

Многомерные преобразования Фурье и их применение в обработке изображений обсуждаются в книгах Гонзалеса (Gonzalez) и Вудса (Woods) [145] и Пратта (Pratt) [279], а в книгах Толимьери (Tolimieri), Эн (An) и Лу (Lu) [336] и Ван Лона (Van Loan) [341] обсуждаются математические аспекты многомерных БПФ.

Изобретение БПФ в середине 1960-х годов часто связывают с работой Кули (Coolet) и Таки (Tukey) [75]. На самом деле БПФ неоднократно изобреталось до этого, но важность его в полной мере не осознавалась до появления современных компьютеров. Хотя Пресс, Тукольски, Веттерлинг и Фланнери приписывают открытие данного метода Рунге (Runge) и Кёнигу (König) в 1924 году, Хейдеман (Heideman), Джонсон (Johnson) и Баррус (Burrus) [162] в своей статье утверждают, что БПФ открыл еще К.Ф. Гаусс (C.F. Gauss) в 1805 г.

Фриго (Frigo) и Джонсон (Johnson) [116] разработали быструю и гибкую реализацию БПФ, названную FFTW (fastest Fourier transform in the West — быстрейшее преобразование Фурье на Западе). FFTW разработано для ситуаций, когда требуется многократное вычисление ДПФ для задач одного и того же размера. Перед тем как приступить к фактическому вычислению ДПФ, FFTW вызывает "планировщик", который путем ряда пробных запусков определяет, как наилучшим образом выполнить декомпозицию для данного размера задачи на конкретной машине. FFTW адаптировано для эффективного использования аппаратного кеша, а когда подзадачи оказываются достаточно малы, FFTW решает их с помощью оптимизированного линейного кода. Кроме того, FFTW обладает необычным преимуществом — временем работы $\Theta(n \lg n)$ для любого размера задачи n, даже когда n представляет собой большое простое число.

Хотя стандартное преобразование Фурье предполагает, что входные данные представляют собой равномерно распределенные по временно́й области точки, другие методики могут аппроксимировать БПФ на "неравноудаленных" данных. Обзор таких методов можно найти в статье Вэра (Ware) [346].

Глава 31. Теоретико-числовые алгоритмы

Ранее теория чисел рассматривалась как элегантная, но почти бесполезная область чистой математики, но в наши дни теоретико-числовые алгоритмы нашли широкое применение, в большей степени благодаря изобретению криптографических схем, основанных на больших простых числах. Применимость этих схем базируется на том, что можно легко находить большие простые числа, а их безопасность — на неизвестности эффективного способа разложения на множители произведения больших простых чисел (или решения других связанных задач, таких как вычисление дискретных логарифмов). В этой главе представлены некоторые вопросы теории чисел и связанные с ними алгоритмы, лежащие в основе таких приложений.

В разделе 31.1 изложены базовые концепции теории чисел, такие как делимость, равенство по модулю и однозначность разложения на множители. В разделе 31.2 исследуется один из самых старых алгоритмов: алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя двух целых чисел. В разделе 31.3 представлен обзор концепций модульной арифметики. В разделе 31.4 изучается множество чисел, которые являются кратными заданного числа a по модулю n, и показано, как с помощью алгоритма Евклида найти все решения уравнения $ax \equiv b \pmod{n}$. В разделе 31.5 вы познакомитесь с китайской теоремой об остатках. В разделе 31.6 рассматриваются степени заданного числа a по модулю n, а также представлен алгоритм повторного возведения в квадрат (repeated-squaring algorithm) для эффективного вычисления величины $a^b \bmod n$ для заданных a, b и n. Эта операция является ключевой при эффективной проверке простоты чисел и для многих современных криптографических схем. В разделе 31.7 описывается криптографическая система с открытым ключом RSA. В разделе 31.8 исследуется рандомизированный тест простоты чисел, с помощью которого можно организовать эффективный поиск больших простых чисел, что является важной задачей при создании ключей для криптографической системы RSA. Наконец в разделе 31.9 представлен обзор простого, но эффективного эвристического подхода к задаче о разложении небольших целых чисел на множители. Интересно, что задача факторизации — одна из тех, которой, возможно, лучше было бы оставаться трудноразрешимой, поскольку от того, до какой степени трудно раскладывать на множители большие целые числа, зависит надежность RSA-кодирования.

Размер входных данных и стоимость арифметических вычислений

Поскольку нам предстоит работать с большими целыми числами, следует уточнить, что́ будет подразумеваться под размером входных данных и под стоимостью элементарных арифметических операций.

В этой главе "больши́е входные данные" — это обычно входные данные, содержащие "большие целые числа", а не "большое количество целых чисел" (как это было в задаче сортировки). Таким образом, размер входных данных будет определяться *количеством битов*, необходимых для представления этих данных, а не просто количеством чисел в них. Время работы алгоритма, на вход которого подаются целые числа $a_1, a_2, \ldots, a_k$, является ***полиномиальным*** (polynomial), если он выполняется за время, которое выражается полиномиальной функцией от величин $\lg a_1, \lg a_2, \ldots, \lg a_k$, т.е. если это время является полиномиальным от длины входных величин в бинарном представлении.

В большей части настоящей книги было удобно считать элементарные арифметические операции (умножение, деление или вычисление остатка) примитивными, т.е. такими, которые выполняются за единичный промежуток времени. При таком предположении в качестве основы для разумной оценки фактического времени работы алгоритма на компьютере может выступать количество перечисленных арифметических операций, выполняющихся в алгоритме. Однако если входные числа достаточно большие, то для выполнения элементарных операций может потребоваться значительное время, и удобнее измерять сложность теоретико-числового алгоритма количеством ***битовых операций*** (bit operation). В этой модели для перемножения двух β-битовых целых чисел обычным методом необходимо $\Theta(\beta^2)$ битовых операций. Аналогично операцию деления β-битового целого числа на более короткое целое число или операцию вычисления остатка от деления β-битового целого числа на более короткое целое число с помощью простого алгоритма можно выполнить за время $\Theta(\beta^2)$ (см. упр. 31.1.12). Известны и более быстрые методы. Например, время перемножения двух β-битовых целых чисел методом декомпозиции равно $\Theta(\beta^{\lg 3})$, а время работы самого быстрого из известных на сегодняшний день методов равно $\Theta(\beta \lg \beta \lg \lg \beta)$. Однако на практике зачастую наилучшим оказывается алгоритм со временем работы $\Theta(\beta^2)$, и наш анализ будет основываться именно на этой оценке.

В общем случае алгоритмы в этой главе анализируются как в терминах количества арифметических операций, так и в терминах количества битовых операций, требуемых для их выполнения.

31.1. Элементарные понятия теории чисел

В этом разделе приведен краткий обзор понятий, которые используются в элементарной теории чисел, изучающей свойства множества целых чисел $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ и множества натуральных[1] чисел $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Делимость и делители

Основным в теории чисел является понятие делимости одного целого числа на другое. Обозначение $d \mid a$ (читается как "d ***делит*** a" или "a ***делится на*** d") подразумевает, что для некоторого целого числа k выполняется равенство $a = kd$. Число 0 делится на все целые числа. Если $a > 0$ и $d \mid a$, то $|d| \leq |a|$. Если $d \mid a$, то говорят также, что a — ***кратно*** (multiple) d. Если a не делится на d, то пишут $d \nmid a$.

Если $d \mid a$ и $d \geq 0$, то говорят, что d — ***делитель*** (divisor) числа a. Заметим, что $d \mid a$ тогда и только тогда, когда $-d \mid a$, поэтому без потери общности делители можно определить как неотрицательные числа, подразумевая, что число, противоположное по знаку любому делителю числа a, также является делителем числа a. Делитель числа a, отличного от нуля, лежит в пределах от 1 до $|a|$. Например, делителями числа 24 являются числа 1, 2, 3, 4, 6, 8, 12 и 24.

Каждое целое число a делится на ***тривиальные делители*** (trivial divisors) 1 и a. Нетривиальные делители числа a называются также ***множителями*** (factors) a. Например, множители числа 20 — 2, 4, 5 и 10.

Простые и составные числа

Целое число $a > 1$, единственными делителями которого являются тривиальные делители 1 и a, называют ***простым числом*** (prime number) или просто ***простым***. Простые числа обладают многими особыми свойствами и играют важную роль в теории чисел. Ниже приведено 20 первых простых чисел в порядке возрастания:

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67,\ 71\ .$$

В упр. 31.1.2 предлагается доказать, что простых чисел бесконечно много. Целое число $a > 1$, которое не является простым, называется ***составным*** (composite). Например, число 39 — составное, поскольку $3 \mid 39$. Целое число 1 называют ***единицей*** (unit), и оно не является ни простым, ни составным. Аналогично целое число 0 и все отрицательные целые числа также не являются ни простыми, ни составными.

---

[1] Определение натуральных чисел в американской литературе отличается от определения натуральных чисел в отечественной математике, где $\mathbb{N} = \{1, 2, \ldots\}$. В данной главе мы будем использовать определение натуральных чисел из оригинального издания книги. — *Примеч. ред.*

Теорема о делении, остатки и равенство по модулю

Относительно заданного числа n все целые числа можно разбить на те, которые являются кратными числу n, и те, которые не являются кратными числу n. Бóльшая часть теории чисел основана на уточнении этого деления, в котором последняя категория чисел классифицируется в соответствии с тем, чему равен остаток их деления на n. В основе этого уточнения лежит сформулированная ниже теорема (ее доказательство здесь не приводится, но его можно найти, например, в учебнике Найвена (Niven) и Цукермана (Zuckerman) [263]).

Теорема 31.1 (Теорема о делении)

Для любого целого a и любого положительного целого n существует единственная пара целых чисел q и r, таких, что $0 \leq r < n$ и $a = qn + r$. ■

Величина $q = \lfloor a/n \rfloor$ называется ***частным*** (quotient) деления. Величина $r = a \bmod n$ называется ***остатком*** (reminder или residue) деления. Таким образом, $n \mid a$ тогда и только тогда, когда $a \bmod n = 0$.

В соответствии с тем, чему равны остатки чисел по модулю n, их можно разбить на n классов эквивалентности. ***Класс эквивалентности по модулю*** $\boldsymbol{n}$ (equivalence class modulo n), в котором содержится целое число a, имеет вид

$$[a]_n = \{a + kn : k \in \mathbb{Z}\} .$$

Например, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; другие обозначения этого множества — $[-4]_7$ и $[10]_7$. Используя обозначения на с. 79, можно сказать, что запись $a \in [b]_n$ — это то же самое, что и запись $a \equiv b \pmod n$. Множество всех таких классов эквивалентности имеет вид

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} . \tag{31.1}$$

Когда вы видите определение

$$\mathbb{Z}_n = \{0, 1, \ldots, n - 1\} , \tag{31.2}$$

вы должны читать его как эквивалентное уравнению (31.1), понимая, что 0 представляет $[0]_n$, 1 представляет $[1]_n$ и т.д.; каждый класс представлен своим наименьшим неотрицательным элементом. Однако при этом следует иметь в виду именно соответствующие классы эквивалентности. Например, -1 в качестве члена класса $\mathbb{Z}_n$ обозначает $[n-1]_n$, поскольку $-1 \equiv n - 1 \pmod n$.

Общие делители и наибольшие общие делители

Если d является делителем числа a, а также делителем числа b, то d представляет собой общий делитель чисел a и b. Например, делителями числа 30 являются 1, 2, 3, 5, 6, 10, 15 и 30, так что общими делителями чисел 24 и 30 являются 1, 2, 3 и 6. Заметим, что 1 — общий делитель двух любых целых чисел.

Важное свойство общих делителей заключается в том, что

$$\text{из}\quad d \mid a \text{ и } d \mid b \quad \text{следует}\quad d \mid (a+b) \text{ и } d \mid (a-b)\ . \tag{31.3}$$

В более общем виде можно записать, что для всех целых x и y

$$\text{из}\quad d \mid a \text{ и } d \mid b \quad \text{следует}\quad d \mid (ax+by)\ . \tag{31.4}$$

Кроме того, если $a \mid b$, то либо $|a| \le |b|$, либо $b = 0$, откуда вытекает, что

$$\text{из}\quad a \mid b \text{ и } b \mid a \quad \text{следует}\quad a = \pm b\ . \tag{31.5}$$

Наибольший общий делитель (greatest common divisor) двух целых чисел a и b, не являющихся одновременно нулем, — это самый большой из общих делителей чисел a и b; он обозначается как $\gcd(a,b)$[2]. Например, $\gcd(24,30) = 6$, $\gcd(5,7) = 1$ и $\gcd(0,9) = 9$. Если числа a и b отличны от нуля, то значение $\gcd(a,b)$ находится в интервале от 1 до $\min(|a|,|b|)$. Величину $\gcd(0,0)$ считаем равной нулю; это необходимо для универсальной применимости стандартных свойств функции gcd (как, например, в уравнении (31.9) ниже).

Ниже перечислены элементарные свойства функции gcd:

$$\gcd(a,b) = \gcd(b,a)\ , \tag{31.6}$$
$$\gcd(a,b) = \gcd(-a,b)\ , \tag{31.7}$$
$$\gcd(a,b) = \gcd(|a|,|b|)\ , \tag{31.8}$$
$$\gcd(a,0) = |a|\ , \tag{31.9}$$
$$\gcd(a,ka) = |a| \quad \text{для любого } k \in \mathbb{Z}\ . \tag{31.10}$$

Сформулированная ниже теорема дает полезную альтернативную характеристику функции $\gcd(a,b)$.

Теорема 31.2

Если a и b — произвольные целые числа, не равные одновременно нулю, то величина $\gcd(a,b)$ равна наименьшему положительному элементу множества $\{ax+by : x,y \in \mathbb{Z}\}$ линейных комбинаций чисел a и b.

Доказательство. Обозначим через s наименьшую положительную из таких линейных комбинаций чисел a и b, и пусть $s = ax + by$ при некоторых $x, y \in \mathbb{Z}$. Пусть также $q = \lfloor a/s \rfloor$. Тогда из уравнения (3.8) следует

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax+by) \\ &= a\,(1-qx) + b\,(-qy)\ , \end{aligned}$$

---

[2]В отечественной литературе применяется также обозначение НОД(a, b). — *Примеч. пер.*

поэтому величина $a \bmod s$ также является линейной комбинацией чисел a и b. Но поскольку $0 \le a \bmod s < s$, выполняется соотношение $a \bmod s = 0$, так как s — наименьшая положительная из таких линейных комбинаций. Поэтому $s \mid a$; аналогично можно доказать, что $s \mid b$. Таким образом, s является общим делителем чисел a и b, поэтому справедливо неравенство $\gcd(a, b) \ge s$. Из уравнения (31.4) следует, что $\gcd(a, b) \mid s$, поскольку величина $\gcd(a, b)$ делит и a, и b, а s представляет собой линейную комбинацию этих чисел. Но из того, что $\gcd(a, b) \mid s$ и $s > 0$, следует соотношение $\gcd(a, b) \le s$. Совместное применение неравенств $\gcd(a, b) \ge s$ и $\gcd(a, b) \le s$ доказывает справедливость равенства $\gcd(a, b) = s$; следовательно, можно сделать вывод, что s — наибольший общий делитель чисел a и b. ■

Следствие 31.3

Для любых целых чисел a и b из соотношений $d \mid a$ и $d \mid b$ следует $d \mid \gcd(a, b)$.

Доказательство. Это следствие является результатом уравнения (31.4), поскольку согласно теореме 31.2 величина $\gcd(a, b)$ является линейной комбинацией чисел a и b. ■

Следствие 31.4

Для любых целых чисел a и b и произвольного неотрицательного целого числа n справедливо соотношение

$$\gcd(an, bn) = n \gcd(a, b) \ .$$

Доказательство. Если $n = 0$, доказательство тривиально. Если $n > 0$, то $\gcd(an, bn)$ является наименьшим положительным элементом множества $\{anx + bny : x, y \in \mathbb{Z}\}$, в n раз превышающим наименьший положительный элемент множества $\{ax + by : x, y \in \mathbb{Z}\}$. ■

Следствие 31.5

Для всех положительных целых чисел n, a и b из $n \mid ab$ и $\gcd(a, n) = 1$ следует, что $n \mid b$.

Доказательство. Доказательство этого следствия читателю предлагается выполнить самостоятельно в упр. 31.1.5. ■

Взаимно простые целые числа

Два целых числа a и b называются ***взаимно простыми*** (relatively prime), если единственный их общий делитель равен 1, т.е. если $\gcd(a, b) = 1$. Например, числа 8 и 15 взаимно простые, поскольку делители числа 8 равны 1, 2, 4 и 8, а делители числа 15 — 1, 3, 5 и 15. В сформулированной ниже теореме утверждается, что если два целых числа взаимно простые с целым числом p, то их произведение также является взаимно простым с числом p.

Теорема 31.6

Для любых целых чисел a, b и p из $\gcd(a,p) = 1$ и $\gcd(b,p) = 1$ следует $\gcd(ab,p) = 1$.

Доказательство. Из теоремы 31.2 следует, что существуют такие целые числа x, y, x' и y', для которых

$$ax + py = 1 \ ,$$
$$bx' + py' = 1 \ .$$

Перемножив эти уравнения и перегруппировав слагаемые, получаем

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 \ .$$

Поскольку положительная линейная комбинация чисел ab и p равна 1, применение теоремы 31.2 завершает доказательство. ■

Говорят, что целые числа $n_1, n_2, \ldots, n_k$ ***попарно взаимно простые*** (pairwise relatively prime), если при $i \neq j$ выполняется соотношение $\gcd(n_i, n_j) = 1$.

Единственность разложения на множители

Элементарный, но важный факт, касающийся делимости на простые числа, сформулирован в приведенной ниже теореме.

Теорема 31.7

Для любого простого числа p и всех целых чисел a и b из условия $p \mid ab$ следуют либо соотношение $p \mid a$, либо соотношение $p \mid b$, либо они оба.

Доказательство. Проведем доказательство методом от противного. Предположим, что $p \mid ab$, но $p \nmid a$ и $p \nmid b$. Таким образом, $\gcd(a,p) = 1$ и $\gcd(b,p) = 1$, поскольку единственными делителями числа p являются 1 и p и согласно предположению на p не делится ни a, ни b. Тогда из теоремы 31.6 следует, что $\gcd(ab,p) = 1$, а это противоречит условию, что $p \mid ab$, поскольку из него следует, что $\gcd(ab,p) = p$. Это противоречие и завершает доказательство теоремы. ■

Из теоремы 31.7 следует, что любое целое число можно единственным образом представить в виде произведения простых чисел.

Теорема 31.8 (Единственность разложения на множители)

Имеется единственный способ представления составного числа a в виде произведения

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \ ,$$

где p_i — простые числа, $p_1 < p_2 < \cdots < p_r$, а e_i — положительные целые числа.

Доказательство. Доказательство этой теоремы оставляется читателю в качестве упр. 31.1.11. ■

В качестве примера число 6000 можно разложить на простые числа следующим единственным образом: $2^4 \cdot 3 \cdot 5^3$.

Упражнения

31.1.1

Докажите, что если $a > b > 0$ и $c = a + b$, то $c \bmod a = b$.

31.1.2

Докажите, что простых чисел бесконечно много. (*Указание:* покажите, что ни одно из простых чисел $p_1, p_2, \ldots, p_k$ не является делителем числа $(p_1 p_2 \cdots p_k) + 1$.)

31.1.3

Докажите, что если $a \mid b$ и $b \mid c$, то $a \mid c$.

31.1.4

Докажите, что если p простое и $0 < k < p$, то $\gcd(k, p) = 1$.

31.1.5

Докажите следствие 31.5.

31.1.6

Докажите, что если p простое и $0 < k < p$, то $p \mid \binom{p}{k}$. Сделайте вывод о том, что для всех целых чисел a и b и всех простых p

$$(a + b)^p \equiv a^p + b^p \pmod{p} .$$

31.1.7

Докажите, что если a и b являются произвольными положительными целыми числами, такими, что $a \mid b$, то

$$(x \bmod b) \bmod a = x \bmod a$$

для любого x. Докажите, что при тех же исходных предположениях

$$\text{из } x \equiv y \pmod{b} \quad \text{следует} \quad x \equiv y \pmod{a}$$

для любых целых чисел x и y.

31.1.8

Говорят, что для всех целых $k > 0$ целое число n представляет собой ***k*-ю степень** (*k*th power), если существует целое число a, такое, что $a^k = n$. Кроме того, число $n > 1$ является ***нетривиальной степенью*** (nontrivial power), если оно является k-й степенью для некоторого целого $k > 1$. Покажите, как определить, является ли заданное β-битовое целое число n нетривиальной степенью, за время, выражаемое полиномиальной функцией от β.

31.1.9

Докажите уравнения (31.6)–(31.10).

31.1.10

Покажите, что оператор gcd ассоциативен, т.е. докажите, что для всех целых чисел a, b и c

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) .$$

31.1.11 ★

Докажите теорему 31.8.

31.1.12

Разработайте эффективный алгоритм для операций деления β-битового целого числа на более короткое целое число и для вычисления остатка от деления β-битового целого числа на более короткое целое число. Алгоритм должен выполняться за время $\Theta(\beta^2)$.

31.1.13

Разработайте эффективный алгоритм преобразования заданного β-битового (двоичного) целого числа в десятичное. Докажите, что если умножение или деление целых чисел, длина которых не превышает β, выполняется за время $M(\beta)$, то преобразование из двоичной в десятичную систему счисления можно выполнить за время $\Theta(M(\beta)\lg\beta)$. (*Указание:* воспользуйтесь стратегией "разделяй и властвуй", при которой верхняя и нижняя половины результата получаются путем отдельного выполнения рекурсивных процедур.)

31.2. Наибольший общий делитель

В этом разделе рассматривается алгоритм Евклида, предназначенный для эффективного вычисления наибольшего общего делителя двух целых чисел. Анализ времени работы этого алгоритма выявляет удивительную связь с числами Фибоначчи, которые являются наихудшими входными данными для этого алгоритма.

Ограничимся в этом разделе только неотрицательными целыми числами. Это ограничение обосновывается уравнением (31.8), которое гласит, что $\gcd(a, b) = \gcd(|a|, |b|)$.

В принципе, можно вычислить $\gcd(a, b)$ для положительных целых чисел a и b, пользуясь их разложением на простые множители. Действительно, если

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.11}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \tag{31.12}$$

где нулевые показатели экспоненты используются для того, чтобы в обоих разложениях были представлены одинаковые множества простых чисел $p_1, p_2, \ldots, p_r$,

то можно показать (см. упр. 31.2.1), что

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} . \tag{31.13}$$

Однако в разделе 31.9 будет показано, что время работы самых эффективных на сегодняшний день алгоритмов разложения на множители не выражается полиномиальной функцией; они работают менее производительно. Таким образом, непохоже, чтобы этот подход к вычислению наибольшего общего делителя привел к эффективному алгоритму.

Алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя, основан на сформулированной ниже теореме.

Теорема 31.9 (Рекурсивная теорема о наибольшем общем делителе)

Для любого неотрицательного целого числа a и любого положительного целого числа b справедливо соотношение

$$\gcd(a, b) = \gcd(b, a \bmod b) .$$

Доказательство. Покажем, что величины $\gcd(a, b)$ и $\gcd(b, a \bmod b)$ делятся одна на другую, а поэтому согласно (31.5) они должны быть равны одна другой (поскольку обе они неотрицательны).

Сначала покажем, что $\gcd(a, b) \mid \gcd(b, a \bmod b)$. Если положить $d = \gcd(a, b)$, то $d \mid a$ и $d \mid b$. Согласно (3.8) $a \bmod b = a - qb$, где $q = \lfloor a/b \rfloor$. Поскольку $a \bmod b$ оказывается линейной комбинацией a и b, из уравнения (31.4) вытекает, что $d \mid (a \bmod b)$. Таким образом, поскольку $d \mid b$ и $d \mid (a \bmod b)$, из следствия 31.3 вытекает, что $d \mid \gcd(b, a \bmod b)$ или, что эквивалентно, что

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \tag{31.14}$$

Соотношение $\gcd(b, a \bmod b) \mid \gcd(a, b)$ доказывается почти так же. Если мы теперь положим $d = \gcd(b, a \bmod b)$, то $d \mid b$ и $d \mid (a \bmod b)$. Поскольку $a = qb + (a \bmod b)$, где $q = \lfloor a/b \rfloor$, получается, что a является линейной комбинацией b и $(a \bmod b)$. Согласно (31.4) заключаем, что $d \mid a$. Поскольку $d \mid b$ и $d \mid a$, согласно следствию 31.3 справедливо $d \mid \gcd(a, b)$ или, что эквивалентно,

$$\gcd(b, a \bmod b) \mid \gcd(a, b). \tag{31.15}$$

Использование уравнения (31.5) в комбинации с уравнениями (31.14) и (31.15) завершает доказательство. ■

Алгоритм Евклида

В книге Евклида "*Начала*" (ок. 300 г. до н.э.) описывается приведенный ниже алгоритм вычисления gcd (хотя на самом деле он может иметь и более раннее происхождение). Алгоритм Евклида выражается в виде рекурсивной программы и основан непосредственно на теореме 31.9. В качестве входных данных в нем выступают произвольные неотрицательные целые числа a и b.

```
EUCLID(a, b)
1  if b == 0
2      return a
3  else return EUCLID(b, a mod b)
```

В качестве примера работы процедуры EUCLID рассмотрим вычисление величины $\gcd(30, 21)$:

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3 \,. \end{aligned}$$

В приведенных выше выкладках мы видим три рекурсивных вызова процедуры EUCLID.

Корректность процедуры EUCLID следует из теоремы 31.9, а также из того факта, что если алгоритм в строке 2 возвращает значение a, то $b = 0$, так что из (31.9) следует, что $\gcd(a, b) = \gcd(a, 0) = a$. Работа этого рекурсивного алгоритма не может продолжаться до бесконечности, поскольку второй аргумент процедуры всегда строго убывает при каждом рекурсивном вызове, и он всегда неотрицательный. Таким образом, алгоритм EUCLID всегда завершается и дает правильный ответ.

Время работы алгоритма Евклида

Проанализируем наихудшее время работы алгоритма EUCLID как функцию размера входных данных a и b. Без потери общности предположим, что $a > b \geq 0$. Это предположение легко обосновать, если заметить, что если $b > a \geq 0$, то в процедуре EUCLID(a, b) сразу же выполняется рекурсивный вызов процедуры EUCLID(b, a). Другими словами, если первый аргумент меньше второго, то алгоритм EUCLID затрачивает один дополнительный вызов на перестановку аргументов и продолжает свою работу. Аналогично, если $b = a > 0$, то процедура завершается после одного рекурсивного вызова, поскольку $a \bmod b = 0$.

Полное время работы алгоритма EUCLID пропорционально количеству выполняемых им рекурсивных вызовов. В нашем анализе используются числа Фибоначчи F_k, определяемые рекуррентным соотношением (3.22).

Лемма 31.10

Если $a > b \geq 1$ и в результате вызова процедуры EUCLID(a, b) выполняется $k \geq 1$ рекурсивных вызовов, то $a \geq F_{k+2}$ и $b \geq F_{k+1}$.

Доказательство. Докажем лемму путем индукции по k. В качестве базиса индукции примем $k = 1$. Тогда $b \geq 1 = F_2$ и, поскольку $a > b$, автоматически выполняется соотношение $a \geq 2 = F_3$. Поскольку $b > (a \bmod b)$, при каждом рекурсивном вызове первый аргумент строго больше второго; таким образом, предположение $a > b$ выполняется при каждом рекурсивном вызове.

Примем в качестве гипотезы индукции, что лемма справедлива, если произведено $k - 1$ рекурсивных вызовов; затем докажем, что она выполняется, если произведено k рекурсивных вызовов. Поскольку $k > 0$, то и $b > 0$, и в процедуре EUCLID(a, b) рекурсивно вызывается процедура EUCLID$(b, a \bmod b)$, в которой, в свою очередь, делается $k - 1$ рекурсивных вызовов. Далее, согласно гипотезе индукции выполняются неравенства $b \geq F_{k+1}$ (что доказывает часть леммы) и $(a \bmod b) \geq F_k$. Мы имеем

$$\begin{aligned} b + (a \bmod b) &= b + (a - b\lfloor a/b \rfloor) \\ &\leq a , \end{aligned}$$

поскольку из $a > b > 0$ следует $\lfloor a/b \rfloor \geq 1$. Таким образом,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2} . \end{aligned}$$

■

Из этой леммы непосредственно следует сформулированная ниже теорема.

Теорема 31.11 (Теорема Ламе (Lamé))

Если для произвольного целого числа $k \geq 1$ выполняются условия $a > b \geq 1$ и $b < F_{k+1}$, то в вызове процедуры EUCLID(a, b) производится менее k рекурсивных вызовов. ■

Можно показать, что верхняя граница теоремы 31.11 является лучшей из возможных, показав, что вызов EUCLID(F_{k+1}, F_k) выполняет ровно $k - 1$ рекурсивных вызовов при $k \geq 2$. Воспользуемся индукцией по k. В базисном случае $k = 2$, и вызов EUCLID(F_3, F_2) делает ровно один рекурсивный вызов EUCLID$(1, 0)$. (Мы начинаем с $k = 2$, поскольку при $k = 1$ не выполняется неравенство $F_2 > F_1$.) В качестве шага индукции примем, что EUCLID(F_k, F_{k-1}) выполняет ровно $k-2$ рекурсивных вызова. При $k > 2$ мы имеем $F_k > F_{k-1} > 0$ и $F_{k+1} = F_k + F_{k-1}$, и согласно упр. 31.1.1 $F_{k+1} \bmod F_k = F_{k-1}$. Таким образом,

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, F_{k+1} \bmod F_k) \\ &= \gcd(F_k, F_{k-1}) . \end{aligned}$$

Следовательно, в процедуре EUCLID(F_{k+1}, F_k) рекурсия осуществляется на один раз больше, чем в процедуре EUCLID(F_k, F_{k-1}), или ровно $k-1$ раз, что совпадает с верхней границей теоремы 31.11.

Поскольку число F_k приблизительно равно $\phi^k/\sqrt{5}$, где ϕ — золотое сечение $(1 + \sqrt{5})/2$, определенное уравнением (3.24), количество рекурсивных вызовов в процедуре EUCLID равно $O(\lg b)$. (Более точная оценка предлагается в упр. 31.2.5.) Отсюда следует, что если с помощью алгоритма EUCLID обрабатывается два β-битовых числа, то в нем выполняется $O(\beta)$ арифметических операций и $O(\beta^3)$ битовых операций (в предположении, что при умножении и де-

лении β-битовых чисел выполняется $O(\beta^2)$ битовых операций). Справедливость этой оценки предлагается показать в задаче 31.2.

Развернутая форма алгоритма Евклида

Теперь перепишем алгоритм Евклида так, чтобы с его помощью можно было извлекать дополнительную полезную информацию, в частности, чтобы можно было вычислять целые коэффициенты x и y, такие, что

$$d = \gcd(a, b) = ax + by\ . \tag{31.16}$$

Заметим, что числа x и y могут быть равными нулю или отрицательными. Эти коэффициенты окажутся полезными позже при вычислении модульных мультипликативных обратных значений. В качестве входных данных процедуры EXTENDED-EUCLID выступает пара неотрицательных целых чисел, а на выходе эта процедура возвращает тройку чисел (d, x, y), удовлетворяющих уравнению (31.16).

```
EXTENDED-EUCLID(a, b)
1  if b == 0
2      return (a, 1, 0)
3  else (d', x', y') = EXTENDED-EUCLID(b, a mod b)
4      (d, x, y) = (d', y', x' − ⌊a/b⌋ y')
5      return (d, x, y)
```

На рис. 31.1 проиллюстрировано вычисление gcd(99, 78) процедурой EXTENDED-EUCLID.

Процедура EXTENDED-EUCLID представляет собой разновидность процедуры EUCLID. Строка 1 в ней эквивалентна проверке "$b == 0$" в строке 1 процедуры EUCLID. Если $b = 0$, то процедура EXTENDED-EUCLID в строке 2 возвращает не только значение $d = a$, но и коэффициенты $x = 1$ и $y = 0$, так что $a = ax + by$. Если $b \neq 0$, то процедура EXTENDED-EUCLID сначала вычисляет набор величин

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

Рис. 31.1. Вычисление gcd(99, 78) процедурой EXTENDED-EUCLID. В каждой строке показан один уровень рекурсии: значения входных данных a и b, вычисленное значение $\lfloor a/b \rfloor$ и возвращаемые значения d, x и y. Возвращаемая тройка (d, x, y) становится тройкой (d', x', y'), используемой на следующем более высоком уровне рекурсии. Вызов EXTENDED-EUCLID(99, 78) возвращает $(3, -11, 14)$, так что $\gcd(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

(d', x', y'), таких, что $d' = \gcd(b, a \bmod b)$ и

$$d' = bx' + (a \bmod b)y' . \quad (31.17)$$

Как и в процедуре EUCLID, в этом случае мы имеем $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. Чтобы получить значения x и y, для которых выполняется равенство $d = ax + by$, сначала перепишем уравнение (31.17) с использованием равенств $d = d'$ и (3.8):

$$\begin{aligned} d &= bx' + (a - b\lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') . \end{aligned}$$

Таким образом, выбор $x = y'$ и $y = x' - \lfloor a/b \rfloor y'$ удовлетворяет уравнению $d = ax + by$, что доказывает корректность процедуры EXTENDED-EUCLID.

Поскольку количество рекурсивных вызовов в процедуре EUCLID равно количеству рекурсивных вызовов в процедуре EXTENDED-EUCLID, время работы процедуры EUCLID с точностью до постоянного множителя равно времени работы процедуры EXTENDED-EUCLID. Другими словами, при $a > b > 0$ количество рекурсивных вызовов равно $O(\lg b)$.

Упражнения

31.2.1

Докажите, что из уравнений (31.11) и (31.12) вытекает уравнение (31.13).

31.2.2

Вычислите значения (d, x, y), возвращаемые при вызове процедуры EXTENDED-EUCLID(899, 493).

31.2.3

Докажите, что для всех целых чисел a, k и n справедливо соотношение

$$\gcd(a, n) = \gcd(a + kn, n) .$$

31.2.4

Перепишите алгоритм EUCLID в итеративном виде, с использованием памяти фиксированного объема (т.е. в ней должно храниться не более некоторого фиксированного количества целочисленных значений).

31.2.5

Покажите, что если $a > b \geq 0$, то вызов EUCLID(a, b) делает не более $1 + \log_\phi b$ рекурсивных вызовов. Улучшите эту оценку до $1 + \log_\phi(b/\gcd(a, b))$.

31.2.6

Что возвращает вызов EXTENDED-EUCLID(F_{k+1}, F_k)? Докажите верность своего ответа.

31.2.7

Определим функцию gcd для более чем двух аргументов с помощью рекурсивного уравнения $\gcd(a_0, a_1, \ldots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \ldots, a_n))$. Покажите, что значение этой функции не зависит от порядка ее аргументов. Покажите также, как найти целые числа $x_0, x_1, \ldots, x_n$, такие, что $\gcd(a_0, a_1, \ldots, a_n) = a_0 x_0 + a_1 x_1 + \cdots + a_n x_n$. Покажите, что количество операций деления, выполняемых алгоритмом, равно $O(n + \lg(\max\{a_0, a_1, \ldots, a_n\}))$.

31.2.8

Определим ***наименьшее общее кратное*** (least common multiple) n целых чисел $a_1, a_2, \ldots, a_n$, обозначаемое $\mathrm{lcm}(a_1, a_2, \ldots, a_n)$, как наименьшее неотрицательное целое число, кратное каждому из аргументов a_i. Покажите, как эффективно вычислить величину $\mathrm{lcm}(a_1, a_2, \ldots, a_n)$, используя в качестве подпрограммы (двухаргументную) операцию gcd.

31.2.9

Докажите, что числа n_1, n_2, n_3 и n_4 попарно взаимно простые тогда и только тогда, когда

$$\gcd(n_1 n_2, n_3 n_4) = \gcd(n_1 n_3, n_2 n_4) = 1 .$$

Покажите, что справедливо более общее утверждение, согласно которому числа $n_1, n_2, \ldots, n_k$ попарно взаимно просты тогда и только тогда, когда $\lceil \lg k \rceil$ пар чисел, образованных из n_i, являются взаимно простыми.

31.3. Модульная арифметика

Неформально модульную арифметику можно считать обычной арифметикой, вычисления в которой выполняются над целыми числами, за исключением того, что если мы работаем по модулю числа n, то любой результат x заменяется элементом множества $\{0, 1, \ldots, n-1\}$, равным числу x по модулю n (т.е. x заменяется величиной $x \bmod n$). Описанной выше неформальной модели достаточно, если ограничиться операциями сложения, вычитания и умножения. Более формализованная модель модульной арифметики, которая будет представлена ниже, лучше всего описывается в рамках теории групп.

Конечные группы

Группа (group) $(S, \oplus)$ представляет собой множество S, на котором определена бинарная операция $\oplus$, обладающая перечисленными ниже свойствами.

1. **Замкнутость:** для всех a, $b \in S$ справедливо $a \oplus b \in S$.
2. **Существование единицы:** существует элемент $e \in S$, называемый ***единичным*** (identity) элементом группы, такой, что $e \oplus a = a \oplus e = a$ для всех $a \in S$.
3. **Ассоциативность:** для всех a, b, $c \in S$ выполняется $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

4. **Существование обратного элемента:** для каждого $a \in S$ существует единственный элемент $b \in S$, называемый ***обратным*** (inverse) к a, такой, что $a \oplus b = b \oplus a = e$.

В качестве примера рассмотрим уже знакомую нам группу $(\mathbb{Z}, +)$ целых чисел $\mathbb{Z}$ с операцией сложения: ее единичный элемент — 0, а обратный элемент к любому числу a — число $-a$. Если группа $(S, \oplus)$ обладает ***свойством коммутативности*** (commutative law) $a \oplus b = b \oplus a$ для всех $a, b \in S$, то это ***абелева группа*** (abelian group). Если группа $(S, \oplus)$ удовлетворяет условию $|S| < \infty$, т.е. количество ее элементов конечно, то она называется ***конечной*** (finite group).

Группы, определяемые сложением и умножением по модулю

С помощью операций сложения и умножения по модулю n, где n — положительное целое число, можно образовать две конечные абелевы группы. Эти группы основаны на классах эквивалентности целых чисел по модулю n, определенных в разделе 31.1.

Чтобы определить группу над $\mathbb{Z}_n$, нужно задать подходящие бинарные операции, полученные путем переопределения обычных операций сложения и умножения. Операции сложения и умножения для $\mathbb{Z}_n$ определить легко, поскольку классы эквивалентности двух целых чисел однозначно определяют класс эквивалентности их суммы или произведения. Другими словами, если $a \equiv a' \pmod{n}$ и $b \equiv b' \pmod{n}$, то

$$a + b \equiv a' + b' \pmod{n} ,$$
$$ab \equiv a'b' \pmod{n} .$$

Таким образом, мы определяем сложение и умножение по модулю n, обозначаемые как $+_n$ и $\cdot_n$, следующим образом:

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n , \\ [a]_n \cdot_n [b]_n &= [ab]_n . \end{aligned} \tag{31.18}$$

(Вычитание для $\mathbb{Z}_n$ можно легко определить по аналогии, как $[a]_n -_n [b]_n = [a - b]_n$, однако с делением, как мы сможем вскоре убедиться, дело обстоит сложнее.) Эти факты подтверждают удобную общепринятую практику использования наименьшего неотрицательного элемента каждого класса эквивалентности как представительного при вычислениях в $\mathbb{Z}_n$. Сложение, вычитание и умножение представителей классов выполняется как обычно, но затем каждый результат x заменяется соответствующим представителем его класса эквивалентности (т.е. величиной $x \bmod n$).

На основе определения операции сложения по модулю n определяется ***аддитивная группа по модулю n*** (additive group modulo n) $(\mathbb{Z}_n, +_n)$. Размер аддитивной группы по модулю n равен $|\mathbb{Z}_n| = n$. На рис. 31.2, (а) представлены результаты выполнения операций для группы $(\mathbb{Z}_6, +_6)$.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(а)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(б)

Рис. 31.2. Две конечные группы. Классы эквивалентности указаны их представительными элементами. **(а)** Группа $(\mathbb{Z}_6, +_6)$. **(б)** Группа $(\mathbb{Z}_{15}^*, \cdot_{15})$.

Теорема 31.12

Система $(\mathbb{Z}_n, +_n)$ является конечной абелевой группой.

Доказательство. Из уравнения (31.18) следует замкнутость $(\mathbb{Z}_n, +_n)$. Ассоциативность и коммутативность $+_n$ следует из ассоциативности и коммутативности $+$:

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n \\ &= [(a+b)+c]_n \\ &= [a+(b+c)]_n \\ &= [a]_n +_n [b+c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n) ,\end{aligned}$$

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n \\ &= [b+a]_n \\ &= [b]_n +_n [a]_n .\end{aligned}$$

Единичным элементом $(\mathbb{Z}_n, +_n)$ является 0 (т.е. $[0]_n$). Элементом, (аддитивно) обратным к элементу a (т.е. к $[a]_n$), является элемент $-a$ (т.е. $[-a]_n$ или $[n-a]_n$), поскольку $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

На основе операции умножения по модулю n определяется ***мультипликативная группа по модулю n*** (multiplicative group modulo n) $(\mathbb{Z}_n^*, \cdot_n)$. Элементы этой группы образуют множество $\mathbb{Z}_n^*$, образованное из элементов множества $\mathbb{Z}_n$, взаимно простых с n, так что каждый из них имеет единственный обратный к нему элемент по модулю n:

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \gcd(a, n) = 1\} .$$

Чтобы убедиться в том, что группа $\mathbb{Z}_n^*$ вполне определена, заметим, что для $0 \leq a < n$ при всех целых k выполняется соотношение $a \equiv (a + kn) \pmod n$. Поэтому из $\gcd(a, n) = 1$ согласно результатам упр. 31.2.3 для всех целых k следует, что $\gcd(a + kn, n) = 1$. Поскольку $[a]_n = \{a + kn : k \in \mathbb{Z}\}$, множество $\mathbb{Z}_n^*$ вполне определено. Примером такой группы является множество

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

в качестве групповой операции в которой выступает операция умножения по модулю 15. (Здесь элемент $[a]_{15}$ обозначается как a; например, элемент $[7]_{15}$ обозначается как 7.) На рис. 31.2, (б) показана группа $(\mathbb{Z}_{15}^*, \cdot_{15})$. Например, $8 \cdot 11 \equiv 13 \pmod{15}$. Единичным элементом в этой группе является 1.

Теорема 31.13

Система $(\mathbb{Z}_n^*, \cdot_n)$ является конечной абелевой группой.

Доказательство. Из теоремы 31.6 вытекает замкнутость $(\mathbb{Z}_n^*, \cdot_n)$. Ассоциативность и коммутативность можно доказать для $\cdot_n$ так же, как это было сделано для $+_n$ в доказательстве теоремы 31.12. Единичным элементом является $[1]_n$. Чтобы показать существование обратных элементов, предположим, что a является элементом $\mathbb{Z}_n^*$, а процедура EXTENDED-EUCLID(a, n) возвращает (d, x, y). Тогда $d = 1$, поскольку $a \in \mathbb{Z}_n^*$, и

$$ax + ny = 1 \tag{31.19}$$

или, что эквивалентно,

$$ax \equiv 1 \pmod n .$$

Таким образом, $[x]_n$ является мультипликативным обратным к $[a]_n$ по модулю n. Кроме того, $[x]_n \in \mathbb{Z}_n^*$. Чтобы увидеть, почему это так, заметим, что (31.19) указывает, что наименьшая положительная линейная комбинация x и n должна быть равна 1. Следовательно, из теоремы 31.2 вытекает, что $\gcd(x, n) = 1$. Доказательство того, что обратные элементы определяются однозначно, отложим до рассмотрения следствия 31.26. ■

В качестве примера вычисления мультипликативных обратных элементов рассмотрим случай $a = 5$ и $n = 11$. Процедура EXTENDED-EUCLID(a, n) возвращает $(d, x, y) = (1, -2, 1)$, так что $1 = 5 \cdot (-2) + 11 \cdot 1$. Таким образом, $[-2]_{11}$ (т.е. $[9]_{11}$) является мультипликативным обратным к $[5]_{11}$.

Далее, в оставшейся части этой главы, когда речь будет идти о группах $(\mathbb{Z}_n, +_n)$ и $(\mathbb{Z}_n^*, \cdot_n)$, мы будем следовать обычной удобной практике обозначения классов эквивалентности представляющими их элементами, а операций $+_n$ и $\cdot_n$ — знаками обычных арифметических операций $+$ и $\cdot$ (последний знак может опускаться, так что $ab = a \cdot b$) соответственно. Кроме того, эквивалентность по модулю n можно интерпретировать как равенство в $\mathbb{Z}_n$. Например, оба выражения,

приведенные ниже, обозначают одно и то же:

$$ax \equiv b \pmod{n} ,$$
$$[a]_n \cdot_n [x]_n = [b]_n .$$

Для дальнейшего удобства иногда группа $(S, \oplus)$ будет обозначаться просто как S, а из контекста будет понятно, какая именно операция подразумевается. Таким образом, группы $(\mathbb{Z}_n, +_n)$ и $(\mathbb{Z}_n^*, \cdot_n)$ будут обозначаться как $\mathbb{Z}_n$ и $\mathbb{Z}_n^*$ соответственно.

Элемент, обратный (относительно умножения) к элементу a, будет обозначаться как $(a^{-1} \bmod n)$. Операция деления в группе $\mathbb{Z}_n^*$ определяется уравнением $a/b \equiv ab^{-1} \pmod{n}$. Например, в $\mathbb{Z}_{15}^*$ мы имеем $7^{-1} \equiv 13 \pmod{15}$, поскольку $7 \cdot 13 = 91 \equiv 1 \pmod{15}$, так что $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

Обозначим размер $\mathbb{Z}_n^*$ как $\phi(n)$. Эта функция, известная как ***ϕ-функция Эйлера***, удовлетворяет уравнению

$$\phi(n) = n \prod_{p\,:\,p \text{ простое и } p \mid n} \left(1 - \frac{1}{p}\right) , \tag{31.20}$$

где индекс p пробегает значения всех простых делителей числа n (включая само n, если оно простое). Здесь мы не станем доказывать эту формулу, а попробуем дать для нее интуитивно понятное пояснение. Выпишем все возможные остатки от деления на n — $\{0, 1, \ldots, n-1\}$, а затем для каждого простого делителя p числа n вычеркнем из этого списка все элементы, кратные p. Например, простые делители числа 45 — числа 3 и 5, поэтому

$$\begin{aligned}\phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 .\end{aligned}$$

Если p — простое число, то $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$, и

$$\begin{aligned}\phi(p) &= p\left(1 - \frac{1}{p}\right) \\ &= p - 1 .\end{aligned} \tag{31.21}$$

Если n — составное, то $\phi(n) < n - 1$; можно показать, что

$$\phi(n) > \frac{n}{e^{\gamma} \ln\ln n + \frac{3}{\ln\ln n}} \tag{31.22}$$

для $n \geq 3$, где $\gamma = 0.5772156649\ldots$ — ***константа Эйлера***. Немного более простая (и более неточная) нижняя граница для $n > 5$ имеет вид

$$\phi(n) > \frac{n}{6\ln\ln n} . \tag{31.23}$$

Нижняя граница (31.22), по сути, является наилучшей возможной, поскольку

$$\liminf_{n\to\infty} \frac{\phi(n)}{n/\ln\ln n} = e^{-\gamma} . \tag{31.24}$$

Подгруппы

Если $(S, \oplus)$ является группой, $S' \subseteq S$ и $(S', \oplus)$ также является группой, то $(S', \oplus)$ представляет собой ***подгруппу*** $(S, \oplus)$. Например, четные целые числа образуют подгруппу целых чисел относительно операции сложения. Приведенная далее теорема служит полезным инструментом распознавания подгрупп.

Теорема 31.14 (Непустое замкнутое подмножество конечной группы является подгруппой)

Если $(S, \oplus)$ является конечной группой, а S' — любое непустое подмножество S, такое, что $a \oplus b \in S'$ для всех $a, b \in S'$, то $(S', \oplus)$ является подгруппой $(S, \oplus)$.

Доказательство. Доказать эту теорему читателю предлагается самостоятельно в упр. 31.3.3. ■

Например, множество $\{0, 2, 4, 6\}$ образует подгруппу группы $\mathbb{Z}_8$, поскольку оно непустое и замкнуто относительно операции $+$ (т.е. оно замкнуто относительно операции $+_8$).

Сформулированная ниже теорема крайне полезна для оценки размера подгруппы; доказательство этой теоремы опущено.

Теорема 31.15 (Теорема Лагранжа)

Если $(S, \oplus)$ представляет собой конечную группу и $(S', \oplus)$ является подгруппой $(S, \oplus)$, то $|S'|$ является делителем $|S|$. ■

Подгруппу S' группы S называют ***истинной*** (proper) подгруппой, если $S' \neq S$. Приведенное ниже следствие из этой теоремы окажется полезным при анализе теста простых чисел Миллера–Рабина (Miller–Rabin), описанного в разделе 31.8.

Следствие 31.16

Если S' является истинной подгруппой конечной группы S, то $|S'| \leq |S|/2$. ■

Подгруппы, сгенерированные элементом группы

Теорема 31.14 предоставляет интересный способ, позволяющий сформировать подгруппу конечной группы $(S, \oplus)$. Для этого следует выбрать какой-нибудь элемент группы a и выделить все элементы, которые можно сгенерировать с помо-

щью элемента a и групповой операции. Точнее говоря, определим элемент $a^{(k)}$ для $k \geq 1$ как

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k} .$$

Например, если взять $a = 2$ из группы $\mathbb{Z}_6$, то последовательность $a^{(1)}, a^{(2)}, a^{(3)}, \ldots$ имеет вид

$$2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots .$$

В группе $\mathbb{Z}_n$ мы имеем $a^{(k)} = ka \bmod n$, а в группе $\mathbb{Z}_n^*$ — $a^{(k)} = a^k \bmod n$. Определим ***подгруппу, сгенерированную элементом*** a (обозначается как $\langle a \rangle$ или $(\langle a \rangle, \oplus)$) следующим образом:

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

Будем говорить, что a ***генерирует*** подгруппу $\langle a \rangle$ или что a является ***генератором*** $\langle a \rangle$. Поскольку множество S конечно, $\langle a \rangle$ является конечным подмножеством S, возможно, включающим все элементы S. Так как из ассоциативности $\oplus$ вытекает

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ замкнуто, а значит, согласно теореме 31.14 $\langle a \rangle$ является подгруппой S. Например, в $\mathbb{Z}_6$ мы имеем

$$\begin{aligned} \langle 0 \rangle &= \{0\} , \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\} , \\ \langle 2 \rangle &= \{0, 2, 4\} . \end{aligned}$$

Аналогично в $\mathbb{Z}_7^*$ мы имеем

$$\begin{aligned} \langle 1 \rangle &= \{1\} , \\ \langle 2 \rangle &= \{1, 2, 4\} , \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\} . \end{aligned}$$

Порядок (order) элемента a (в группе S), записываемый как $\mathrm{ord}(a)$, определяется как наименьшее положительное целое число t, такое, что $a^{(t)} = e$.

Теорема 31.17

Для любой конечной группы $(S, \oplus)$ и любого ее элемента $a \in S$ порядок элемента a равен размеру генерируемой им подгруппы, т.е. $\mathrm{ord}(a) = |\langle a \rangle|$.

Доказательство. Пусть $t = \mathrm{ord}(a)$. Поскольку $a^{(t)} = e$ и $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ для $k \geq 1$, при $i > t$ для некоторого $j < i$ выполняется равенство $a^{(i)} = a^{(j)}$. Таким образом, при генерации элементом a после элемента $a^{(t)}$ не появляется никаких новых элементов. Таким образом, $\langle a \rangle = \{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\}$ и $|\langle a \rangle| \leq t$. Чтобы показать, что $|\langle a \rangle| \geq t$, покажем, что все элементы последовательности

$a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ различны. Будем действовать методом от противного. Предположим, что при некоторых i и j, удовлетворяющих неравенству $1 \leq i < j \leq t$, выполняется равенство $a^{(i)} = a^{(j)}$. Тогда при $k \geq 0$ имеем $a^{(i+k)} = a^{(j+k)}$. Однако из этого следует, что $a^{(i+(t-j))} = a^{(j+(t-j))} = e$. Так мы приходим к противоречию, поскольку $i + (t - j) < t$, но t — наименьшее положительное значение, такое, что $a^{(t)} = e$. Поэтому все элементы последовательности $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ различны, и $|\langle a \rangle| \geq t$. Таким образом, мы приходим к заключению, что $\mathrm{ord}(a) = |\langle a \rangle|$. ■

Следствие 31.18

Последовательность $a^{(1)}, a^{(2)}, \ldots$ периодична с периодом $t = \mathrm{ord}(a)$; т.е. $a^{(i)} = a^{(j)}$ тогда и только тогда, когда $i \equiv j \pmod{t}$. ■

С приведенным следствием согласуется определение $a^{(0)}$ как e, а $a^{(i)}$ для всех целых i — как $a^{(i \bmod t)}$, где $t = \mathrm{ord}(a)$.

Следствие 31.19

Если $(S, \oplus)$ — конечная группа с единичным элементом e, то для всех $a \in S$ выполняется соотношение

$$a^{(|S|)} = e .$$

Доказательство. Из теоремы Лагранжа (теорема 31.15) следует, что $\mathrm{ord}(a) \mid |S|$, так что $|S| \equiv 0 \pmod{t}$, где $t = \mathrm{ord}(a)$. Следовательно, $a^{(|S|)} = a^{(0)} = e$. ■

Упражнения

31.3.1

Составьте таблицы операций для групп $(\mathbb{Z}_4, +_4)$ и $(\mathbb{Z}_5^*, \cdot_5)$. Покажите, что эти группы изоморфны. Для этого продемонстрируйте взаимное однозначное соответствие α между элементами этих групп, такое, что $a + b \equiv c \pmod{4}$ тогда и только тогда, когда $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

31.3.2

Перечислите все подгруппы группы $\mathbb{Z}_9$ и группы $\mathbb{Z}_{13}^*$.

31.3.3

Докажите теорему 31.14.

31.3.4

Покажите, что если p — простое число, а e — положительное целое число, то

$$\phi(p^e) = p^{e-1}(p-1) .$$

31.3.5

Покажите, что для любого целого $n > 1$ и для любого $a \in \mathbb{Z}_n^*$ функция $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$, определенная как $f_a(x) = ax \bmod n$, представляет собой перестановку $\mathbb{Z}_n^*$.

31.4. Решение модульных линейных уравнений

Теперь рассмотрим задачу о решении уравнений вида

$$ax \equiv b \pmod{n} , \tag{31.25}$$

где $a > 0$ и $n > 0$. Эта задача имеет несколько приложений. Например, она используется как часть процедуры, предназначенной для поиска ключей для криптографической схемы RSA с открытым ключом, описанной в разделе 31.7. Предположим, что для заданных чисел a, b и n нужно найти все значения переменной x, которые удовлетворяют уравнению (31.25). Количество решений может быть равным нулю, единице, или быть больше единицы.

Обозначим через $\langle a \rangle$ подгруппу группы $\mathbb{Z}_n$, сгенерированную элементом a. Поскольку $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, уравнение (31.25) имеет решение тогда и только тогда, когда $[b] \in \langle a \rangle$. Теорема Лагранжа (теорема 31.15) говорит о том, что значение $|\langle a \rangle|$ должно быть делителем n. Сформулированная ниже теорема дает точную характеристику подгруппы $\langle a \rangle$.

Теорема 31.20

Для всех положительных целых чисел a и n из $d = \gcd(a, n)$ следует, что

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\} \tag{31.26}$$

в $\mathbb{Z}_n$, так что

$$|\langle a \rangle| = n/d .$$

Доказательство. Начнем с того, что покажем, что $d \in \langle a \rangle$. Вспомним, что $\text{Extended-Euclid}(a, n)$ дает целые числа x' и y', такие, что $ax' + ny' = d$. Таким образом, $ax' \equiv d \pmod{n}$, так что $d \in \langle a \rangle$. Другими словами, d кратно a из $\mathbb{Z}_n$.

Поскольку $d \in \langle a \rangle$, отсюда следует, что каждое кратное d принадлежит $\langle a \rangle$, так как любое кратное кратного a является кратным a. Таким образом, $\langle a \rangle$ содержит все элементы множества $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$, т.е. $\langle d \rangle \subseteq \langle a \rangle$.

Теперь покажем, что $\langle a \rangle \subseteq \langle d \rangle$. Если $m \in \langle a \rangle$, то $m = ax \bmod n$ для некоторого целого x, так что $m = ax + ny$ для некоторого целого y. Однако $d \mid a$ и $d \mid n$, так что $d \mid m$ согласно уравнению (31.4). Следовательно, $m \in \langle d \rangle$.

Объединив полученные результаты, получаем, что $\langle a \rangle = \langle d \rangle$. Чтобы увидеть, что $|\langle a \rangle| = n/d$, заметим, что между 0 и $n - 1$ включительно имеется ровно n/d кратных d. ∎

Следствие 31.21

Уравнение $ax \equiv b \pmod{n}$ разрешимо относительно неизвестного x тогда и только тогда, когда $d \mid b$, где $d = \gcd(a, n)$.

Доказательство. Уравнение $ax \equiv b \pmod{n}$ разрешимо тогда и только тогда, когда $[b] \in \langle a \rangle$, что то же самое, что

$$(b \bmod n) \in \{0, d, 2d, \ldots, ((n/d) - 1)d\}$$

согласно теореме 31.20. Если $0 \leq b < n$, то $b \in \langle a \rangle$ тогда и только тогда, когда $d \mid b$, поскольку эти члены $\langle a \rangle$ являются точными кратными d. Если $b < 0$ или $b \geq n$, следствие вытекает из наблюдения, что $d \mid b$ тогда и только тогда, когда $d \mid (b \bmod n)$, поскольку b и $b \bmod n$ отличаются на кратное n, которое само является кратным d. ■

Следствие 31.22

Уравнение $ax \equiv b \pmod{n}$ либо имеет d различных по модулю n решений, где $d = \gcd(a, n)$, либо не имеет решений вовсе.

Доказательство. Если $ax \equiv b \pmod{n}$ имеет решение, то $b \in \langle a \rangle$. Согласно теореме 31.17 $\mathrm{ord}(a) = |\langle a \rangle|$, так что из следствия 31.18 и теоремы 31.20 вытекает, что последовательность $ai \bmod n$, $i = 0, 1, \ldots$, периодична с периодом $|\langle a \rangle| = n/d$. Если $b \in \langle a \rangle$, то b появляется в последовательности $ai \bmod n$, $i = 0, 1, \ldots, n - 1$, ровно d раз, поскольку блок значений $\langle a \rangle$ длиной (n/d) повторяется ровно d раз при увеличении i от нуля до $n - 1$. Индексы x тех d позиций, для которых $ax \bmod n = b$, и являются решениями уравнения $ax \equiv b \pmod{n}$. ■

Теорема 31.23

Пусть $d = \gcd(a, n)$, и предположим, что $d = ax' + ny'$ для некоторых целых чисел x' и y' (например, вычисленных процедурой EXTENDED-EUCLID). Если $d \mid b$, то уравнение $ax \equiv b \pmod{n}$ имеет в качестве одного из решений значение x_0, где

$$x_0 = x'(b/d) \bmod n \; .$$

Доказательство. Мы имеем

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \quad (\text{поскольку } ax' \equiv d \pmod{n}) \\ &\equiv b \pmod{n} \; , \end{aligned}$$

и, таким образом, x_0 является решением $ax \equiv b \pmod{n}$. ■

Теорема 31.24

Предположим, что уравнение $ax \equiv b \pmod{n}$ разрешимо (т.е. $d \mid b$, где $d = \gcd(a, n)$), и что x_0 является некоторым решением этого уравнения. Тогда это уравнение имеет ровно d различных по модулю n решений, задаваемых соотношением $x_i = x_0 + i(n/d)$ при $i = 0, 1, \ldots, d - 1$.

Доказательство. Поскольку $n/d > 0$ и $0 \leq i(n/d) < n$ для $i = 0, 1, \ldots, d - 1$, все значения $x_0, x_1, \ldots, x_{d-1}$ различны по модулю n. Поскольку x_0 является ре-

шением $ax \equiv b \pmod{n}$, мы имеем $ax_0 \bmod n \equiv b \pmod{n}$. Таким образом, для $i = 0, 1, \ldots, d-1$ имеем

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \quad \text{(так как из } d \mid a \text{ следует, что } ain/d \text{ кратно } n\text{)} \\ &\equiv b \pmod{n} , \end{aligned}$$

а следовательно, $ax_i \equiv b \pmod{n}$, что делает решением также и x_i. Согласно следствию 31.22 уравнение $ax \equiv b \pmod{n}$ имеет ровно d решений, так что все $x_0, x_1, \ldots, x_{d-1}$ должны быть таковыми. ■

Теперь у нас имеется математический аппарат, необходимый для решения уравнения вида $ax \equiv b \pmod{n}$, а ниже приведен алгоритм, который выводит все его решения. На вход алгоритма подаются произвольные положительные целые числа a и n и произвольное целое число b.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') = EXTENDED-EUCLID(a, n)
2  if d | b
3      x0 = x'(b/d) mod n
4      for i = 0 to d − 1
5          print (x0 + i(n/d)) mod n
6  else print "решений нет"
```

В качестве примера работы этой процедуры рассмотрим уравнение $14x \equiv 30 \pmod{100}$ (здесь $a = 14$, $b = 30$ и $n = 100$). Вызвав в строке 1 процедуру EXTENDED-EUCLID, получаем $(d, x', y') = (2, -7, 1)$. Поскольку $2 \mid 30$, выполняются строки 3–5. В строке 3 вычисляется значение $x_0 = (-7)(15) \bmod 100 = 95$. В цикле в строках 4–5 выводятся два решения уравнения, равные 95 и 45.

Процедура MODULAR-LINEAR-EQUATION-SOLVER работает следующим образом. В строке 1 вычисляются величина $d = \gcd(a, n)$, а также значения x' и y', такие, что $d = ax' + ny'$, что свидетельствует о том, что x' удовлетворяет исходному уравнению $ax' \equiv d \pmod{n}$. Если d не является делителем числа b, то уравнение решений не имеет согласно следствию 31.21. В строке 2 проверяется, делится ли b на d; если не делится, то в строке 6 выводится сообщение об отсутствии решений заданного уравнения. В противном случае в строке 3 в соответствии с теоремой 31.23 вычисляется решение x_0 уравнения $ax \equiv b \pmod{n}$. Если найдено одно решение, то согласно теореме 31.24 остальные $d-1$ решений можно получить путем добавления величин, кратных (n/d), по модулю n. Это и делается в цикле **for** в строках 4 и 5, где с шагом (n/d) выводятся все d решений по модулю n, начиная с x_0.

Процедура MODULAR-LINEAR-EQUATION-SOLVER выполняет $O(\lg n + \gcd(a, n))$ арифметических операций, поскольку процедура EXTENDED-EUCLID

выполняет $O(\lg n)$ арифметических операций, а каждая итерация цикла **for** в строках 4 и 5 выполняет фиксированное количество арифметических операций.

Сформулированные ниже следствия теоремы 31.24 позволяют сделать уточнения, представляющие особый интерес.

Следствие 31.25

Для любого $n > 1$ из $\gcd(a, n) = 1$ следует, что уравнение $ax \equiv b \pmod{n}$ имеет единственное по модулю n решение. ■

Значительный интерес представляет распространенный случай $b = 1$. При этом искомое число x является ***мультипликативным обратным*** (multiplicative inverse) к числу a по модулю n.

Следствие 31.26

Для любого $n > 1$ из $\gcd(a, n) = 1$ следует, что уравнение $ax \equiv 1 \pmod{n}$ имеет единственное по модулю n решение. В противном случае оно не имеет решений. ■

Благодаря следствию 31.26 можно использовать запись $a^{-1} \bmod n$ для мультипликативного обратного к a по модулю n, когда a и n взаимно простые. Если $\gcd(a, n) = 1$, то единственным решением уравнения $ax \equiv 1 \pmod{n}$ является целое число x, возвращаемое процедурой EXTENDED-EUCLID, поскольку из уравнения

$$\gcd(a, n) = 1 = ax + ny$$

следует, что $ax \equiv 1 \pmod{n}$. Таким образом, $a^{-1} \bmod n$ можно эффективно вычислить с помощью процедуры EXTENDED-EUCLID.

Упражнения

31.4.1

Найдите все решения уравнения $35x \equiv 10 \pmod{50}$.

31.4.2

Докажите, что из уравнения $ax \equiv ay \pmod{n}$ вытекает $x \equiv y \pmod{n}$, если $\gcd(a, n) = 1$. Покажите, что условие $\gcd(a, n) = 1$ является необходимым, приведя контрпример с $\gcd(a, n) > 1$.

31.4.3

Рассмотрим следующее изменение строки 3 процедуры MODULAR-LINEAR-EQUATION-SOLVER.

3 $\quad x_0 = x'(b/d) \bmod (n/d)$

Будет ли работать такая видоизмененная процедура? Обоснуйте ответ.

32 Зак. 3726

31.4.4 ★

Пусть p — простое число, а $f(x) \equiv f_0 + f_1x + \cdots + f_tx^t \pmod p$ — полином степени t с коэффициентами f_i, взятыми из $\mathbb{Z}_p$. Мы говорим, что $a \in \mathbb{Z}_p$ является ***нулем*** f, если $f(a) \equiv 0 \pmod p$. Докажите, что если a является нулем f, то $f(x) \equiv (x-a)g(x) \pmod p$ для некоторого полинома $g(x)$ степени $t-1$. Докажите по индукции по t, что если p простое, то полином $f(x)$ степени t может иметь не более t различных по модулю p нулей.

31.5. Китайская теорема об остатках

Около 100 г. н.э. китайский математик Сунь-Цзы (Sun-Tsŭ) решил задачу поиска целых чисел x, которые при делении на 3, 5 и 7 дают остатки 2, 3 и 2 соответственно. Одно из таких решений — $x = 23$, а общее решение имеет вид $23 + 105k$, где k — произвольное целое число. "Китайская теорема об остатках" устанавливает соответствие между системой уравнений по взаимно простым модулям (например, 3, 5 и 7) и уравнением по модулю произведения этих чисел (в данном примере это число 105).

Китайская теорема об остатках имеет два основных применения. Пусть целое число n раскладывается на попарно взаимно простые множители $n = n_1n_2\cdots n_k$. Во-первых, эта теорема играет роль описательной "структурной теоремы", которая описывает структуру $\mathbb{Z}_n$ как идентичную структуре декартового произведения $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ с покомпонентным сложением и умножением по модулю n_i в i-м компоненте. Во-вторых, с помощью этого описания во многих случаях можно разработать эффективные алгоритмы, поскольку работа с каждой из систем $\mathbb{Z}_{n_i}$ может оказаться эффективнее (в терминах битовых операций), чем работа по модулю n.

Теорема 31.27 (Китайская теорема об остатках)

Пусть $n = n_1n_2\cdots n_k$, где n_i — попарно взаимно простые числа. Рассмотрим соответствие

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) , \qquad (31.27)$$

где $a \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$ и

$$a_i = a \bmod n_i$$

при $i = 1, 2, \ldots, k$. Тогда отображение (31.27) является взаимно однозначным отображением (биекцией) между $\mathbb{Z}_n$ и декартовым произведением $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Операции, которые выполняются над элементами $\mathbb{Z}_n$, можно эквивалентно выполнять над соответствующими k-кортежами, независимо выполняя операции над каждым компонентом в соответствующей системе. То есть, если

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) ,$$
$$b \leftrightarrow (b_1, b_2, \ldots, b_k) ,$$

то

$$(a+b) \bmod n \leftrightarrow ((a_1+b_1) \bmod n_1, \ldots, (a_k+b_k) \bmod n_k) , \quad (31.28)$$
$$(a-b) \bmod n \leftrightarrow ((a_1-b_1) \bmod n_1, \ldots, (a_k-b_k) \bmod n_k) , \quad (31.29)$$
$$(ab) \bmod n \leftrightarrow (a_1b_1 \bmod n_1, \ldots, a_kb_k \bmod n_k) . \quad (31.30)$$

Доказательство. Преобразование одного представления в другое осуществляется достаточно прямолинейно. Переход от a к $(a_1, a_2, \ldots, a_k)$ очень простой, и для него требуется выполнить всего k операций "mod".

Вычислить элемент a по заданным входным элементам $(a_1, a_2, \ldots, a_k)$ несколько сложнее, и это можно сделать следующим образом. Сначала определим величины $m_i = n/n_i$ для $i = 1, 2, \ldots, k$; таким образом, m_i представляет собой произведение всех значений n_j, отличных от n_i: $m_i = n_1n_2 \cdots n_{i-1}n_{i+1} \cdots n_k$. Затем определим

$$c_i = m_i(m_i^{-1} \bmod n_i) \quad (31.31)$$

для $i = 1, 2, \ldots, k$. Уравнение (31.31) всегда вполне определено: поскольку числа m_i и n_i взаимно простые (согласно теореме 31.6), следствие 31.26 гарантирует существование величины $m_i^{-1} \bmod n_i$. Наконец величину a можно вычислить как функцию от a_1, a_2, ..., a_k следующим образом:

$$a \equiv (a_1c_1 + a_2c_2 + \cdots + a_kc_k) \pmod{n} . \quad (31.32)$$

Теперь покажем, что уравнение (31.32) гарантирует выполнение соотношения $a \equiv a_i \pmod{n_i}$ для $i = 1, 2, \ldots, k$. Заметим, что если $j \neq i$, то $m_j \equiv 0 \pmod{n_i}$, откуда следует, что $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Заметим также, что из уравнения (31.31) следует $c_i \equiv 1 \pmod{n_i}$. Таким образом, мы получаем красивое и полезное соответствие

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0) ,$$

где все компоненты равны нулю, кроме i-го, который равен единице. Векторы c_i в определенном смысле образуют базис представления. Поэтому для каждого i можно записать

$$\begin{aligned} a &\equiv a_ic_i \pmod{n_i} \\ &\equiv a_im_i(m_i^{-1} \bmod n_i) \pmod{n_i} \\ &\equiv a_i \pmod{n_i} , \end{aligned}$$

что и требовалось показать: представленный метод вычисления величины a по заданным значениям a_i дает результат, удовлетворяющий ограничениям $a \equiv a_i \pmod{n_i}$ для $i = 1, 2, \ldots, k$. Это соответствие взаимно однозначное, поскольку преобразования можно выполнять в обоих направлениях. Наконец уравнения (31.28)–(31.30) непосредственно следуют из результатов упр. 31.1.7, поскольку соотношение $x \bmod n_i = (x \bmod n) \bmod n_i$ справедливо при любом x и $i = 1, 2, \ldots, k$. ■

Сформулируем следствия, которые понадобятся нам в последующих разделах.

Следствие 31.28

Если $n_1, n_2, \ldots, n_k$ попарно взаимно простые и $n = n_1 n_2 \cdots n_k$, то для любых целых чисел $a_1, a_2, \ldots, a_k$ система уравнений

$$x \equiv a_i \pmod{n_i} ,$$

где $i = 1, 2, \ldots, k$, имеет единственное решение по модулю n относительно неизвестной величины x. ■

Следствие 31.29

Если $n_1, n_2, \ldots, n_k$ попарно взаимно простые и $n = n_1 n_2 \cdots n_k$, то для любых целых чисел x и a

$$x \equiv a \pmod{n_i} ,$$

где $i = 1, 2, \ldots, k$, выполняется тогда и только тогда, когда

$$x \equiv a \pmod{n} .$$ ■

В качестве примера применения китайской теоремы об остатках предположим, что дана система уравнений

$$a \equiv 2 \pmod{5} ,$$
$$a \equiv 3 \pmod{13} ,$$

так что $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$ и $n_2 = m_1 = 13$, и необходимо вычислить $a \bmod 65$, поскольку $n = n_1 n_2 = 65$. Так как $13^{-1} \equiv 2 \pmod{5}$ и $5^{-1} \equiv 8 \pmod{13}$, имеем

$$c_1 = 13(2 \bmod 5) = 26 ,$$
$$c_2 = 5(8 \bmod 13) = 40$$

и

$$a \equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65}$$
$$\equiv 52 + 120 \pmod{65}$$
$$\equiv 42 \pmod{65} .$$

Данный пример проиллюстрирован на рис. 31.3.

Таким образом, операции по модулю n можно выполнять непосредственно, а можно перейти к представлению, в котором вычисления проводятся отдельно по модулям n_i, т.е. выбирать более удобный способ. Эти вычисления полностью эквивалентны.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Рис. 31.3. Иллюстрация китайской теоремы об остатках при $n_1 = 5$ и $n_2 = 13$. В этом примере $c_1 = 26$ и $c_2 = 40$. На пересечении строки i и столбца j показано значение a по модулю 65, такое, что $a \bmod 5 = i$ и $a \bmod 13 = j$. Обратите внимание, что в ячейке на пересечении строки 0 и столбца 0 содержится значение 0. Аналогично в ячейке на пересечении строки 4 и столбца 12 содержится 64 (эквивалент -1). Поскольку $c_1 = 26$, смещение вниз вдоль столбца на одну строку приводит к увеличению значения a на 26. Аналогично $c_2 = 40$ означает, что сдвиг вправо вдоль строки на один столбец приводит к увеличению значения a на 40. Увеличение значения a на 1 соответствует сдвигу по диагонали вниз и вправо с переходом по достижении границ из нижней в верхнюю строку и из крайнего справа столбца в крайний слева.

Упражнения

31.5.1

Найдите все решения уравнений $x \equiv 4 \pmod 5$ и $x \equiv 5 \pmod{11}$.

31.5.2

Найдите все целые числа x, которые при делении на 9, 8, 7 дают соответственно остатки 1, 2, 3.

31.5.3

Докажите, что при выполнении условий теоремы 31.27, если $\gcd(a, n) = 1$,

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)) .$$

31.5.4

Докажите, что при выполнении условий теоремы 31.27 для произвольного полинома f количество корней уравнения $f(x) \equiv 0 \pmod n$ равно произведению количества корней каждого из уравнений $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, $\ldots$, $f(x) \equiv 0 \pmod{n_k}$.

31.6. Степени элемента

Мы часто рассматриваем кратные данному элементу a по модулю n; столь же естественно рассматривать и последовательность степеней a по модулю n, где $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \ldots \tag{31.33}$$

по модулю n. Индексация начинается от нуля; нулевой член последовательности равен $a^0 \bmod n = 1$, а i-й член — $a^i \bmod n$. Ниже в качестве примера приведены

степени чисел 3 по модулю 7.

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	$\cdots$

А вот как выглядят степени 2 по модулю 7.

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	$\cdots$

В этом разделе $\langle a \rangle$ будет обозначать подгруппу группы $\mathbb{Z}_n^*$, сгенерированную элементом a путем повторного умножения, а $\operatorname{ord}_n(a)$ (порядок a по модулю n) будет обозначать порядок элемента a в группе $\mathbb{Z}_n^*$. Например, в группе $\mathbb{Z}_7^*$ подгруппа $\langle 2 \rangle = \{1, 2, 4\}$, а $\operatorname{ord}_7(2) = 3$. Используя определение функции Эйлера $\phi(n)$ как размера группы $\mathbb{Z}_n^*$ (см. раздел 31.3), преобразуем следствие 31.19 с использованием обозначений $\mathbb{Z}_n^*$, чтобы получить теорему Эйлера и сформулировать ее для частного случая группы $\mathbb{Z}_p^*$, где p — простое число, что даст нам теорему Ферма.

Теорема 31.30 (Теорема Эйлера)

Для любого целого $n > 1$

$$a^{\phi(n)} \equiv 1 \pmod{n} \quad \text{для всех } a \in \mathbb{Z}_n^* . \qquad \blacksquare$$

Теорема 31.31 (Теорема Ферма)

Если p — простое число, то

$$a^{p-1} \equiv 1 \pmod{p} \quad \text{для всех } a \in \mathbb{Z}_p^* .$$

Доказательство. Согласно (31.21), если p — простое число, $\phi(p) = p - 1$. ■

Теорема Ферма применима к каждому элементу $\mathbb{Z}_p$ за исключением 0, поскольку $0 \notin \mathbb{Z}_p^*$. Однако если p — простое, то $a^p \equiv a \pmod{p}$ для всех $a \in \mathbb{Z}_p$.

Если $\operatorname{ord}_n(g) = |\mathbb{Z}_n^*|$, то каждый элемент группы $\mathbb{Z}_n^*$ является степенью g по модулю n и g представляет собой ***первообразный корень*** (primitive root), или ***генератор*** $\mathbb{Z}_n^*$. Например, 3 является первообразным корнем по модулю 7, но 2 первообразным корнем по модулю 7 не является. Если в группе $\mathbb{Z}_n^*$ имеется первообразный корень, то говорят, что она ***циклическая*** (cyclic). Доказательство сформулированной ниже теоремы, доказанной Нивеном (Niven) и Цукерманом (Zuckerman) в [263], опущено.

Теорема 31.32

Значения $n > 1$, для которых группа $\mathbb{Z}_n^*$ является циклической, представляют собой 2, 4, p^e и $2p^e$, для всех простых $p > 2$ и всех положительных целых e. ■

Если g представляет собой первообразный корень группы $\mathbb{Z}_n^*$, а a — произвольный элемент $\mathbb{Z}_n^*$, то существует z, такое, что $g^z \equiv a \pmod{n}$. Это z называется

дискретным логарифмом (discrete logarithm) или ***индексом*** (index) a по модулю n по основанию g; будем записывать это значение как $\text{ind}_{n,g}(a)$.

Теорема 31.33 (Теорема о дискретном логарифме)

Если g является первообразным корнем $\mathbb{Z}_n^*$, то равенство $g^x \equiv g^y \pmod n$ справедливо тогда и только тогда, когда выполняется $x \equiv y \pmod{\phi(n)}$.

Доказательство. Сначала предположим, что $x \equiv y \pmod{\phi(n)}$. Тогда $x = y + k\phi(n)$ для некоторого целого числа k. Следовательно,

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} \pmod n \\ &\equiv g^y \cdot (g^{\phi(n)})^k \pmod n \\ &\equiv g^y \cdot 1^k \pmod n && \text{(согласно теореме Эйлера)} \\ &\equiv g^y \pmod n \ . \end{aligned}$$

И обратно, предположим, что $g^x \equiv g^y \pmod n$. Поскольку последовательность степеней g генерирует каждый элемент $\langle g \rangle$ и $|\langle g \rangle| = \phi(n)$, из следствия 31.18 вытекает, что последовательность степеней g периодична с периодом $\phi(n)$. Следовательно, если $g^x \equiv g^y \pmod n$, должно выполняться $x \equiv y \pmod{\phi(n)}$. ■

Теперь обратим внимание на квадратные корни из 1 по модулю простой степени. Приведенная ниже теорема окажется полезной в разделе 31.8 при разработке алгоритма для проверки простоты.

Теорема 31.34

Если p представляет собой нечетное простое число и $e \geq 1$, то уравнение

$$x^2 \equiv 1 \pmod{p^e} \tag{31.34}$$

имеет только два решения, а именно — $x = 1$ и $x = -1$.

Доказательство. Уравнение (31.34) эквивалентно уравнению

$$p^e \mid (x-1)(x+1) \ .$$

Поскольку $p > 2$, может выполняться только одно из условий $p \mid (x-1)$ и $p \mid (x+1)$, но не оба одновременно. (В противном случае согласно свойству (31.3) p должно быть делителем их разности $(x+1) - (x-1) = 2$.) Если $p \nmid (x-1)$, то $\gcd(p^e, x-1) = 1$, и согласно следствию 31.5 должно выполняться $p^e \mid (x+1)$, т.е. $x \equiv -1 \pmod{p^e}$. Симметрично, если $p \nmid (x+1)$, то $\gcd(p^e, x+1) = 1$, и из следствия 31.5 вытекает, что $p^e \mid (x-1)$, так что $x \equiv 1 \pmod{p^e}$. Следовательно, либо $x \equiv -1 \pmod{p^e}$, либо $x \equiv 1 \pmod{p^e}$. ■

Число x называется ***нетривиальным квадратным корнем* 1 *по модулю* n**, если справедливо уравнение $x^2 \equiv 1 \pmod n$, но x не эквивалентно ни одному из "тривиальных" квадратных корней по модулю n, — ни 1, ни -1. Например,

6 — нетривиальный квадратный корень из 1 по модулю 35. Приведенное ниже следствие теоремы 31.34 будет использовано в разделе 31.8 при доказательстве корректности процедуры Миллера–Рабина (Miller-Rabin) для проверки простоты чисел.

Следствие 31.35

Если существует нетривиальный квадратный корень из 1 по модулю n, то число n составное.

Доказательство. Будем проводить доказательство от противного. Используем теорему, обратную теореме 31.34: если существует нетривиальный квадратный корень из 1 по модулю n, то n не может быть нечетным простым числом или его степенью. Если $x^2 \equiv 1 \pmod 2$, то $x \equiv 1 \pmod 2$, поэтому все квадратные корни 1 по модулю 2 тривиальны. Таким образом, n не может быть простым. Наконец, чтобы существовал нетривиальный квадратный корень из 1, n должно удовлетворять неравенству $n > 1$. Поэтому число n должно быть составным. ∎

Возведение в степень путем многократного возведения в квадрат

В теоретико-числовых вычислениях часто встречается операция возведения одного числа в степень по модулю другого числа, известная под названием ***модульное возведение в степень*** (modular exponentiation). Другими словами, нам нужно найти эффективный способ вычисления величины $a^b \bmod n$, где a и b — неотрицательные целые числа, а n — положительное целое число. Операция модульного возведения в степень играет важную роль во многих подпрограммах, выполняющих проверку простоты чисел, а также в криптографической системе с открытым ключом RSA. Метод эффективного решения этой задачи, в котором используется бинарное представление числа b, называется методом ***многократного возведения в квадрат*** (repeated squaring).

Пусть $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ — бинарное представление числа b. (То есть бинарное представление имеет длину $k + 1$ бит, b_k — старший, а b_0 — младший биты представления.) Приведенная ниже процедура вычисляет $a^c \bmod n$, где c возрастает от 0 до b путем удвоения и приращения.

```
MODULAR-EXPONENTIATION(a, b, n)
c = 0
d = 1
Пусть ⟨b_k, b_{k-1}, ..., b_0⟩ — бинарное представление b
for i = k downto 0
    c = 2c
    d = (d · d) mod n
    if b_i == 1
        c = c + 1
        d = (d · a) mod n
return d
```

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Рис. 31.4. Результаты работы процедуры MODULAR-EXPONENTIATION по вычислению $a^b \pmod n$, где $a = 7$, $b = 560 = \langle 1000110000 \rangle$ и $n = 561$. В таблице показаны значения после каждой итерации цикла **for**. Окончательный результат равен единице

Важной частью процедуры, объясняющей ее название ("многократное возведение в степень"), является возведение в квадрат в строке 6, выполняемое в каждой итерации. В качестве иллюстрации работы алгоритма рассмотрим пример $a = 7$, $b = 560$ и $n = 561$. В этом случае алгоритм вычисляет последовательность величин по модулю 561, приведенную на рис. 31.4. Полученная в алгоритме последовательность показателей степени содержится в строке c таблицы.

В действительности алгоритм вполне может обойтись без переменной c; она добавлена для того, чтобы можно было сформулировать следующий состоящий из двух частей инвариант цикла.

Непосредственно перед каждой итерацией цикла **for** в строках 4–9

1. значение c совпадает с префиксом $\langle b_k, b_{k-1}, \ldots, b_{i+1} \rangle$ бинарного представления b;
2. $d = a^c \bmod n$.

Используем этот инвариант цикла следующим образом.

Инициализация. Изначально $i = k$, так что префикс $\langle b_k, b_{k-1}, \ldots, b_{i+1} \rangle$ пуст, что соответствует $c = 0$. Кроме того, $d = 1 = a^0 \bmod n$.

Сохранение. Обозначим через c' и d' значения c и d в конце итерации цикла **for**, а значит, значения перед началом следующей итерации. Каждая итерация выполняет $c' = 2c$ (если $b_i = 0$) или $c' = 2c + 1$ (если $b_i = 1$), так что c будет иметь корректное значение перед следующей итерацией. Если $b_i = 0$, то $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$. Если $b_i = 1$, то $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. В любом случае перед началом следующей итерации $d = a^c \bmod n$.

Завершение. В момент завершения $i = -1$. Таким образом, $c = b$, поскольку c имеет значение, равное префиксу $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ бинарного представления b. Следовательно, $d = a^c \bmod n = a^b \bmod n$.

Если входные данные a, b и n представляют собой β-битовые числа, то общее количество требуемых арифметических операций равно $O(\beta)$, а общее количество требуемых битовых операций равно $O(\beta^3)$.

Упражнения

31.6.1

Составьте таблицу, в которой был бы показан порядок каждого элемента группы $\mathbb{Z}_{11}^*$. Выберите наименьший первообразный корень g и вычислите таблицу значений $\mathrm{ind}_{11,g}(x)$ для всех $x \in \mathbb{Z}_{11}^*$.

31.6.2

Разработайте алгоритм модульного возведения в степень, в котором биты числа b проверяются не слева направо, а справа налево.

31.6.3

Считая величину $\phi(n)$ известной, объясните, как с помощью процедуры MODULAR-EXPONENTIATION вычислить величину $a^{-1} \bmod n$ для любого $a \in \mathbb{Z}_n^*$.

31.7. Криптосистема с открытым ключом RSA

Криптографическую систему с открытым ключом можно использовать для шифровки сообщений, которыми обмениваются два партнера, чтобы посторонний, перехвативший зашифрованное сообщение, не смог его расшифровать. Кроме того, криптографическая система с открытым ключом позволяет партнерам добавлять в конце электронных сообщений цифровые подписи. Такая подпись представляет собой электронную версию подписи, поставленной от руки на бумажном документе. Кто угодно без труда может ее проверить, но подделать не сможет никто. Кроме того, при изменении хотя бы одного бита в электронном сообщении оно теряет свою достоверность. Таким образом, цифровая подпись позволяет не только идентифицировать автора сообщения, но и обеспечить целостность его содержимого. Она является прекрасным инструментом для подписанных с помощью электронных средств деловых контрактов, электронных чеков, оформляемых в электронном виде заказов на поставку и других электронных документов, которые необходимо идентифицировать.

Криптографическая система с открытым ключом RSA основана на драматическом различии в том, насколько легко находить большие простые числа и насколько сложно раскладывать на множители произведение двух больших простых чисел. В разделе 31.8 описана эффективная процедура поиска больших простых чисел, а в разделе 31.9 обсуждается проблема разложения больших целых чисел на множители.

Криптографические системы с открытым ключом

В криптографической системе с открытым ключом каждый участник располагает как ***открытым ключом*** (public key), так и ***секретным ключом*** (secret key). Каждый ключ — это часть информации. Например, в криптографической системе RSA каждый ключ состоит из пары целых чисел. Пусть в процессе обмена сооб-

щениями принимают участие Алиса и Борис. Обозначим открытый и секретный ключи Алисы через P_A и S_A, а Бориса — через P_B и S_B.

Каждый участник создает свой открытый и секретный ключ самостоятельно. Секретный ключ каждый из них держит в секрете, а открытые ключи можно сообщать кому угодно и даже публиковать. Фактически часто удобно предполагать, что открытый ключ каждого пользователя доступен в каталоге общего пользования, поэтому любой участник общения может легко получить открытый ключ любого другого участника.

Открытый и секретный ключи определяют функции, применимые к любому сообщению. Обозначим через $\mathcal{D}$ множество допустимых сообщений. Например, $\mathcal{D}$ может быть множеством всех битовых последовательностей конечной длины. В простейшей первоначальной формулировке криптографии с открытым ключом требуется, чтобы открытый и секретный ключи задавали взаимно однозначную функцию, отображающую множество $\mathcal{D}$ само на себя. Обозначим функцию, соответствующую открытому ключу Алисы P_A, как $P_A()$, а функцию, соответствующую ее секретному ключу S_A, — как $S_A()$. Таким образом, функции $P_A()$ и $S_A()$ являются перестановками множества $\mathcal{D}$. Предполагается, что существует эффективный алгоритм вычисления функций $P_A()$ и $S_A()$ по заданным ключам P_A и S_A.

Открытый и секретный ключи каждого участника обмена сообщениями образуют "согласованную пару" в том смысле, что они являются взаимно обратными. Другими словами, для любого сообщения $M \in \mathcal{D}$ выполняются соотношения

$$M = S_A(P_A(M)) , \tag{31.35}$$
$$M = P_A(S_A(M)) . \tag{31.36}$$

Произведя последовательное преобразование сообщения M с помощью ключей P_A и S_A (в любом порядке), мы снова получим сообщение M.

В криптографической системе с открытым ключом существенное обстоятельство заключается в том, чтобы никто, кроме Алисы, не мог вычислить функцию $S_A()$ за приемлемое с практической точки зрения время. Секретность сообщений электронной почты, зашифрованных и отправленных Алисе, а также подлинность цифровых подписей Алисы основываются на предположении о том, что только Алиса способна вычислить функцию $S_A()$. Это требование является причиной, по которой Алиса должна держать в секрете ключ S_A. Если бы она этого не делала, то ее ключ потерял бы секретность и криптографическая система не смогла бы предоставить Алисе упомянутые выше возможности. Предположение о том, что только Алиса может вычислить функцию $S_A()$, должно соблюдаться даже несмотря на то, что каждому известен ключ P_A и каждый может эффективно вычислить функцию $P_A()$, обратную функции $S_A()$. Основная сложность, возникающая при разработке работоспособной криптографической системы с открытым ключом, заключается в создании системы, в которой можно легко найти преобразование $P_A()$, но невозможно (или очень сложно) вычислить соответствующее обратное преобразование $S_A()$. Это условие выглядит угрожающе, но мы увидим, как добиться его выполнения.

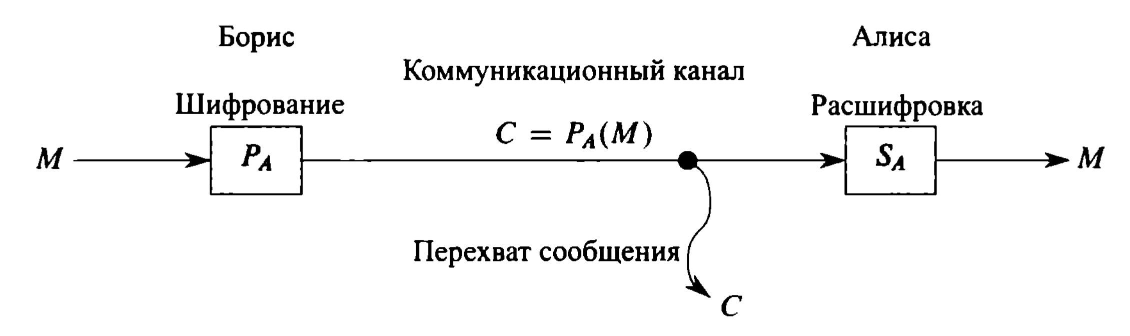


Рис. 31.5. Шифрование в системе с открытым ключом. Борис шифрует сообщение M с помощью открытого ключа Алисы P_A и передает полученное зашифрованное сообщение $C = P_A(M)$ по коммуникационному каналу Алисе. Если кто-то перехватит это сообщение, не получит никакой информации о сообщении M. Алиса получает C и расшифровывает его с помощью своего секретного ключа, получив исходное сообщение $M = S_A(C)$.

В криптографической системе с открытым ключом кодирование проводится так, как показано на рис. 31.5. Предположим, Борис хочет отправить Алисе сообщение M, зашифрованное таким образом, чтобы для постороннего оно выглядело как бессмысленный набор символов. Сценарий отправки сообщения можно представить следующим образом.

- Борис получает открытый ключ Алисы P_A (из каталога общего пользования или непосредственно от Алисы).
- Борис выполняет вычисления $C = P_A(M)$, результатом которых является ***зашифрованный текст*** (ciphertext), соответствующий сообщению M, и отправляет Алисе сообщение C.
- Получив зашифрованное сообщение C, Алиса применяет свой секретный ключ S_A, чтобы преобразовать его в исходное сообщение: $S_A(C) = S_A(P_A(M)) = M$.

Поскольку функции $S_A()$ и $P_A()$ являются обратными по отношению одна к другой, Алиса имеет возможность вычислить сообщение M, располагая сообщением C. Поскольку только Алиса может вычислять функцию $S_A()$, она — единственная, кто способен расшифровать сообщение C. Шифрование сообщения M с помощью функции $P_A()$ защищает его содержимое от всех, кроме Алисы.

Реализовать цифровые подписи в сформулированной модели криптографической системы с открытым ключом столь же просто. (Заметим, что существуют и другие подходы к проблеме создания цифровых подписей, которые здесь не рассматриваются.) Предположим, что Алисе нужно отправить Борису ответ M', подтвержденный цифровой подписью. Сценарий создания цифровой подписи проиллюстрирован на рис. 31.6.

- Алиса вычисляет свою ***цифровую подпись*** (digital signature) σ для сообщения M' с помощью своего секретного ключа S_A и уравнения $\sigma = S_A(M')$.
- Алиса отправляет пару "сообщение/подпись" (M', σ) Борису.
- Получив пару (M', σ), Борис может убедиться, что автор сообщения M' — действительно Алиса, использовав открытый ключ Алисы для проверки равенства $M' = P_A(\sigma)$. (Сообщение M' может содержать имя Алисы, чтобы

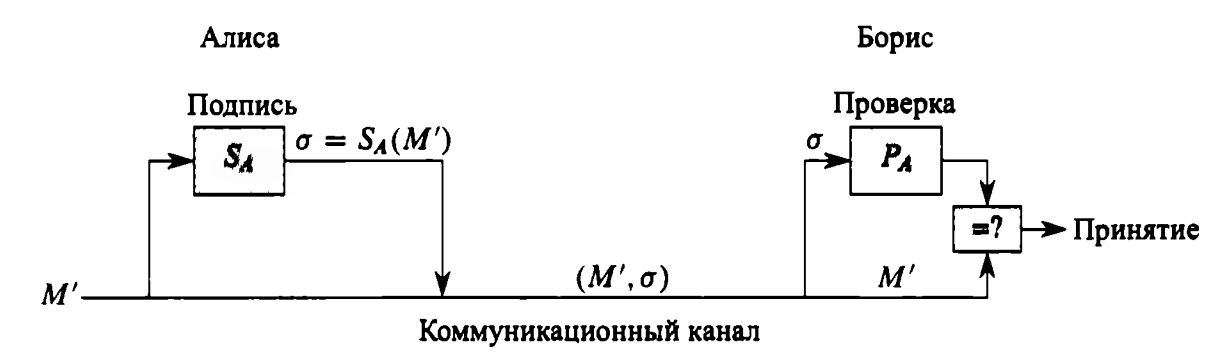


Рис. 31.6. Цифровые подписи в системе с открытым ключом. Алиса подписывает сообщение M', добавляя к нему цифровую подпись $\sigma = S_A(M')$. Она передает пару "сообщение/подпись" (M', σ) Борису, который проверяет сообщение, выясняя, выполняется ли условие $M' = P_A(\sigma)$. Если условие выполнено, он принимает (M', σ) как сообщение, подписанное Алисой.

Борис знал, чей открытый ключ использовать.) Если равенство выполняется, можно смело делать вывод, что сообщение M' действительно подписано Алисой. В противном случае Борис с полным основанием сможет заподозрить, что либо сообщение M' или цифровая подпись σ были искажены в процессе передачи, либо что пара (M', σ) — попытка подлога.

Поскольку цифровая подпись обеспечивает как аутентификацию автора сообщения, так и подтверждение целостности содержимого подписанного сообщения, она служит аналогом подписи, сделанной от руки в конце рукописного документа.

Важное свойство цифровой подписи заключается в том, что каждый, кто имеет доступ к открытому ключу ее автора, должен иметь возможность ее проверить. Один из участников обмена сообщениями после проверки подлинности цифровой подписи может передать подписанное сообщение еще кому-то, кто тоже в состоянии проверить эту подпись. Например, Алиса может переслать Борису в сообщении электронный чек. После того как Борис проверит подпись Алисы на чеке, он может передать его в свой банк, служащие которого также имеют возможность проверить подпись и осуществить соответствующую денежную операцию.

Заметим, что подписанное сообщение не зашифровано; оно пересылается в исходном виде и его содержимое не защищено. Совместно применив приведенные выше схемы, можно создавать сообщения, которые будут и зашифрованными, и содержащими цифровую подпись. Для этого автор должен сначала добавить к сообщению свою цифровую подпись, а затем зашифровать получившуюся в результате пару (состоящую из самого сообщения и подписи к нему) с помощью открытого ключа, принадлежащего получателю. Получатель расшифровывает полученное сообщение с помощью своего секретного ключа. Таким образом, он получит исходное сообщение и цифровую подпись отправителя, которую затем сможет проверить с помощью соответствующего открытого ключа. Если проводить аналогию с пересылкой обычных бумажных документов, то этот процесс похож на то, как если бы автор документа поставил под ним свою подпись, а затем положил его в бумажный конверт и запечатал, чтобы конверт был распечатан только тем человеком, которому адресовано сообщение.

Криптографическая система RSA

В ***криптографической системе с открытым ключом RSA*** (RSA public-key cryptosystem) участники обмена сообщениями создают свои открытые и секретные ключи в соответствии с описанной ниже процедурой.

1. Случайным образом выбираются два больших (скажем, длиной по 1024 бит) простых числа p и q, причем $p \neq q$.
2. Вычисляется $n = pq$.
3. Выбирается маленькое нечетное целое число e, взаимно простое со значением функции $\phi(n)$ (которое согласно уравнению (31.20) равно $(p-1)(q-1)$).
4. Вычисляется число d, мультипликативное обратное к e по модулю $\phi(n)$. (Следствие 31.26 гарантирует, что такое число существует и единственное. Чтобы вычислить d по заданным e и $\phi(n)$, можно воспользоваться методом, описанным в разделе 31.4.)
5. Пара $P = (e, n)$ публикуется в качестве ***открытого ключа RSA*** (RSA pubic key).
6. Пара $S = (d, n)$, играющая роль ***секретного ключа RSA*** (RSA secret key), сохраняется в секрете.

В этой схеме областью определения $\mathcal{D}$ является множество $\mathbb{Z}_n$. Чтобы преобразовать сообщение M с помощью открытого ключа $P = (e, n)$, вычисляем

$$P(M) = M^e \bmod n \; . \tag{31.37}$$

Преобразование зашифрованного текста C с помощью секретного ключа $S = (d, n)$ вычисляется как

$$S(C) = C^d \bmod n \; . \tag{31.38}$$

Эти уравнения применимы для создания как зашифрованных сообщений, так и цифровых подписей. Чтобы создать подпись, ее автор применяет свой секретный ключ не к зашифрованному, а к подписываемому сообщению. Для проверки подписи открытый ключ подписавшегося применяется именно к ней, а не к шифруемому сообщению.

Операции с открытым и секретным ключами можно реализовать с помощью процедуры MODULAR-EXPONENTIATION, описанной в разделе 31.6. Чтобы проанализировать время выполнения этих операций, предположим, что открытый ключ (e, n) и секретный ключ (d, n) удовлетворяют соотношениям $\lg e = O(1)$, $\lg d \leq \beta$ и $\lg n \leq \beta$. Тогда в процессе применения открытого ключа выполняется $O(1)$ умножений по модулю и используется $O(\beta^2)$ битовых операций. В ходе применения секретного ключа выполняется $O(\beta)$ умножений по модулю и используется $O(\beta^3)$ битовых операций.

Теорема 31.36 (Корректность RSA)

Уравнения RSA (31.37) и (31.38) определяют взаимно обратные преобразования множества $\mathbb{Z}_n$, удовлетворяющие соотношениям (31.35) и (31.36).

Доказательство. Из уравнений (31.37) и (31.38) для произвольного $M \in \mathbb{Z}_n$ получаем

$$P(S(M)) = S(P(M)) = M^{ed} \pmod n .$$

Поскольку e и d взаимно обратные относительно умножения по модулю $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

для некоторого целого числа k. Но тогда, если $M \not\equiv 0 \pmod p$, мы имеем

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} \pmod p \\ &\equiv M((M \bmod p)^{p-1})^{k(q-1)} \pmod p \\ &\equiv M(1)^{k(q-1)} \pmod p \qquad \text{(теорема 31.31)} \\ &\equiv M \pmod p . \end{aligned}$$

Кроме того, $M^{ed} \equiv M \pmod p$, если $M \equiv 0 \pmod p$. Таким образом,

$$M^{ed} \equiv M \pmod p$$

для всех M. Аналогично для всех M

$$M^{ed} \equiv M \pmod q .$$

Таким образом, согласно следствию 31.29 из Китайской теоремы об остатках

$$M^{ed} \equiv M \pmod n$$

для всех M. ■

Безопасность криптографической системы RSA в значительной мере основана на сложностях, связанных с разложением больших целых чисел на множители. Если бы кто-нибудь посторонний мог разложить на множители элемент открытого ключа n, то он смог бы использовать множители p и q точно так же, как это сделал создатель открытого ключа, и получить таким образом секретный ключ из открытого. Поэтому, если бы разложение больших целых чисел было простой задачей, так же легко было бы взломать криптографическую систему RSA. Обратное утверждение, которое состоит в том, что сложность взлома схемы RSA обусловлена сложностью разложения больших целых чисел на множители, не доказано. Однако на протяжении двадцатилетних исследований не удалось найти более простого способа взлома, чем тот, при котором необходимо разложить на множители число n. В разделе 31.9 будет показано, что задача разложения больших целых чисел на множители оказалась на удивление сложной. Путем случайного выбора и перемножения между собой двух 1024-битовых простых чисел можно создать открытый ключ, который не удается взломать за приемлемое время с помощью имеющихся в настоящее время технологий. Если в разработке теоретико-числовых алгоритмов не произойдет фундаментального прорыва и если реализовывать

криптографическую схему RSA, придерживаясь рекомендуемых стандартов, то эта схема способна обеспечить высокую степень безопасности в приложениях.

Однако для достижения безопасности в криптографической схеме RSA желательно работать с достаточно большими целыми числами — длиной в сотни или даже более тысячи битов, — чтобы уберечься от возможных достижений в искусстве разложения чисел на множители. Во время написания этой книги (2009) в RSA-модулях обычно использовались числа, длина которых находилась в пределах от 768 до 2048 бит. Для создания модулей таких размеров необходимо иметь возможность эффективно находить большие простые числа. Этой задаче посвящен раздел 31.8.

Для повышения эффективности схема RSA часто используется в "гибридном" режиме совместно с криптографическими системами, не обладающими открытым ключом. В такой системе ключи, предназначенные для зашифровки и расшифровки, идентичны. Если Алисе нужно в частном порядке отправить Борису длинное сообщение M, она выбирает случайный ключ K, предназначенный для быстрой криптографической системы без открытого ключа, и шифрует с его помощью сообщение M. В результате получается зашифрованное сообщение C, длина которого совпадает с длиной сообщения M. При этом ключ K довольно короткий. Затем Алиса шифрует ключ K с помощью открытого RSA-ключа Бориса. Поскольку ключ K короткий, величина $P_B(K)$ вычисляется быстро (намного быстрее, чем величина $P_B(M)$). После этого Алиса пересылает пару $(C, P_B(K))$ Борису, который расшифровывает часть $P_B(K)$, получая в результате ключ K, позволяющий расшифровать сообщение C. В результате этой процедуры Борис получит исходное сообщение M.

Аналогичный гибридный подход часто используется для эффективной генерации цифровых подписей. В этом подходе схема RSA применяется совместно с открытой ***устойчивой к коллизиям хеш-функцией*** h (collision-resistant hash function) — функцией, которую легко вычислить, но для которой нельзя вычислительными средствами найти пару сообщений M и M', удовлетворяющих условию $h(M) = h(M')$. Величина $h(M)$ представляет собой короткий (скажем, 256-битовый) "отпечаток" сообщения M. Если Алиса решит подписать сообщение M, то она сначала применит функцию h к сообщению M, получив в результате отпечаток $h(M)$, который она затем зашифрует с помощью своего секретного ключа. После этого она отправит Борису пару $(M, S_A(h(M)))$ в качестве подписанной версии сообщения M. Чтобы проверить подпись, Борис может вычислить величину $h(M)$ и убедиться, что она совпадает с величиной, полученной в результате применения ключа P_A к полученной компоненте $S_A(h(M))$. Поскольку никто не может создать двух сообщений с одинаковыми отпечатками, невозможно вычислительными средствами изменить подписанное сообщение и в то же время сохранить достоверность подписи.

Наконец заметим, что распространение открытых ключей значительно облегчается благодаря ***сертификатам*** (certificates). Например, предположим, что существует "авторитетный источник" T, открытый ключ которого широко известен. Алиса может получить от T подписанное сообщение (свой сертификат), в котором утверждается, что "P_A — открытый ключ Алисы". Этот сертификат является

"самоаутентифицируемым", поскольку ключ P_T известен всем. В свое подписанное сообщение Алиса может включить свой сертификат, чтобы получатель сразу имел в распоряжении ее открытый ключ и смог проверить ее подпись. Поскольку ключ Алисы подписан источником T, получатель убеждается, что ключ Алисы действительно принадлежит ей.

Упражнения

31.7.1

Рассмотрим набор значений $p = 11$, $q = 29$, $n = 312$ и $e = 3$, образующих ключи в системе RSA. Какое значение d следует использовать в секретном ключе? Как выглядит результат шифровки сообщения $M = 100$?

31.7.2

Докажите, что если показатель степени e в открытом ключе Алисы равен 3 и если постороннему известен показатель степени d секретного ключа Алисы, где $0 < d < \phi(n)$, то он может разложить на множители входящее в состав ключа число n в течение времени, которое выражается полиномиальной функцией от количества битов в числе n. (Читателю может быть интересен тот факт (хотя в упражнении не требуется его доказать), что этот результат остается истинным даже без условия $e = 3$. Дополнительные сведения можно почерпнуть из работы Миллера (Miller) [253].)

31.7.3 ★

Докажите, что схема RSA является мультипликативной в том смысле, что

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n} .$$

Докажите с помощью этого факта, что если злоумышленник располагает процедурой, способной эффективно расшифровать один процент зашифрованных с помощью ключа P_A сообщений из $\mathbb{Z}_n$, то он может воспользоваться вероятностным алгоритмом, чтобы с высокой степенью вероятности расшифровывать все сообщения, зашифрованные с помощью ключа P_A.

★ 31.8. Проверка простоты

В этом разделе рассматривается задача поиска больших простых чисел. Начнем с того, что обсудим плотность распределения простых чисел, после чего перейдем к исследованию правдоподобного (но неполного) подхода к проверке простоты чисел. Затем будет представлен эффективный рандомизированный тест простоты, предложенный Миллером (Miller) и Рабином (Rabin).

Плотность распределения простых чисел

Во многих приложениях, таких, как криптография, возникает необходимость поиска больших "случайных" простых чисел. К счастью, большие простые числа не столь уж редки, поэтому для поиска простого числа путем проверки случайных целых чисел соответствующего размера потребуется не так уж много времени. ***Функция распределения простых чисел*** (prime distribution function) $\pi(n)$ определяется как количество простых чисел, не превышающих числа n. Например, $\pi(10) = 4$, поскольку количество простых чисел, не превышающих 10, равно 4 (это числа 2, 3, 5 и 7). В теореме о распределении простых чисел приводится полезное приближение функции $\pi(n)$.

Теорема 31.37 (Теорема о простых числах)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$$

■

Приближенная оценка $n/\ln n$ дает достаточно точную оценку функции $\pi(n)$ даже при малых n. Например, при $n = 10^9$, когда $\pi(n) = 50\,847\,534$, а $n/\ln n \approx 48\,254\,942$, отклонение не превышает 6%. (Для специалиста в области теории чисел 10^9 — число небольшое.)

Процесс случайного выбора простого числа n и проверку его простоты можно рассматривать как испытания Бернулли (см. раздел В.4). Согласно теореме о простых числах вероятность того, что случайным образом выбранное число n окажется простым, приблизительно равна $1/\ln n$. Геометрическое распределение говорит нам о том, какое количество попыток требуется сделать, чтобы добиться успеха. Согласно (В.32), ожидаемое количество испытаний примерно равно $\ln n$. Таким образом, чтобы найти простое число, длина которого совпадает с длиной числа n, понадобится проверить приблизительно $\ln n$ целых чисел, выбрав их случайным образом в окрестности числа n. Например, следует ожидать, что, чтобы найти 1024-битовое простое число, понадобится перебрать приблизительно $\ln 2^{1024} \approx 710$ случайным образом выбранных 1024-битовых чисел, проверяя их простоту. (Конечно же, ограничившись только нечетными числами, это количество можно уменьшить в два раза.)

В оставшейся части этого раздела рассматривается задача определения того, является ли простым большое целое нечетное число n. Для удобства обозначений предположим, что разложение числа n на простые множители имеет вид

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.39}$$

где $r \geq 1$, $p_1, p_2, \ldots, p_r$ — простые множители числа n, а $e_1, e_2, \ldots, e_r$ — положительные целые числа. Целое число n является простым тогда и только тогда, когда $r = 1$ и $e_1 = 1$.

Одним из элементарных подходов к задаче проверки на простоту является ***пробное деление*** (trial division). При этом предпринимается попытка нацело разделить n на все целые числа $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$. (Здесь также можно опустить все четные числа, большие 2.) Легко понять, что число n простое тогда и только

тогда, когда оно не делится ни на один из пробных множителей. Если предположить, что для обработки каждого пробного делителя требуется фиксированное время, то в худшем случае время решения задачи при таком подходе будет равно $\Theta(\sqrt{n})$, а это означает, что оно выражается экспоненциальной функцией от длины числа n. (Напомним, что если бинарное представление числа n имеет длину β, то $\beta = \lceil \lg(n+1) \rceil$, поэтому $\sqrt{n} = \Theta(2^{\beta/2})$.) Таким образом, пробное деление хорошо работает только при условии, что число n очень мало́ или если окажется, что у него есть маленький простой делитель. Преимущество этого метода заключается в том, что он не только позволяет определить, является ли число n простым, но и находит один из его простых делителей, если оно составное.

В этом разделе нас будет интересовать только факт, является ли заданное число n простым; если оно составное, мы не станем искать его простые множители. Как будет показано в разделе 31.9, разложение чисел на простые множители вычислительными средствами — вычислительно весьма дорогостоящая операция. Возможно, вам покажется странным, что намного легче определить, является ли заданное число n простым, чем разложить составное число на простые множители.

Проверка псевдопростых чисел

Рассмотрим "почти работающий" метод проверки простых чисел, который оказывается вполне пригодным для многих практических приложений. Позже будет представлена усовершенствованная версия этого метода, устраняющая его небольшой дефект. Обозначим через $\mathbb{Z}_n^+$ ненулевые элементы множества $\mathbb{Z}_n$:

$$\mathbb{Z}_n^+ = \{1, 2, \ldots, n-1\} .$$

Если n — простое, то $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

Будем говорить, что число n ***псевдопростое по основанию a*** (base-a pseudoprime), если оно составное и

$$a^{n-1} \equiv 1 \pmod{n} . \tag{31.40}$$

Из теоремы Ферма (теорема 31.31) следует, что если n — простое, то оно удовлетворяет уравнению (31.40) для каждого элемента a множества $\mathbb{Z}_n^+$. Таким образом, если удается найти какой-нибудь элемент $a \in \mathbb{Z}_n^+$, такой, что n *не* удовлетворяет уравнению (31.40), то n — определенно составное. Удивительно, что обратное утверждение *почти* справедливо, поэтому этот критерий является почти идеальным тестом на простоту. Мы проверяем, удовлетворяет ли число n уравнению (31.40) при $a = 2$. Если это не так, то можно сделать вывод, что n — составное. В противном случае можно выдвинуть гипотезу, что n — простое (в то время как фактически известно лишь то, что оно либо простое, либо псевдопростое по основанию 2).

Приведенная ниже процедура проверяет число n на простоту описанным способом. В ней используется процедура MODULAR-EXPONENTIATION, описанная в разделе 31.6. Предполагается, что входное значение n — нечетное целое число, большее 2.

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n − 1, n) ≢ 1 (mod n)
2      return СОСТАВНОЕ      // Гарантированно
3  else return ПРОСТОЕ       // Будем надеяться!
```

Эта процедура может допускать ошибки, но только одного типа. Если процедура говорит, что n — составное, то это всегда верно. Если же она утверждает, что n — простое, то это заключение ошибочно только тогда, когда n — псевдопростое по основанию 2.

Как часто эта процедура допускает ошибки? На удивление редко. Всего имеется лишь 22 значения n, меньших 10 000, для которых ответ будет неправильным. Первые четыре таких значения равны 341, 561, 654 и 1105. Можно показать, что вероятность того, что в этой процедуре будет допущена ошибка для случайно выбранного β-битового числа, стремится к нулю при $\beta \to \infty$. Воспользовавшись результатами статьи Померанца (Pomerance) [277], содержащей более точную оценку количества псевдопростых чисел фиксированного размера по основанию 2, можно заключить, что случайным образом выбранное 512-битовое число, названное приведенной выше процедурой простым, окажется псевдопростым по основанию 2 менее чем в одном случае из 10^{20}, а случайным образом выбранное 1024-битовое число, названное простым, окажется псевдопростым по основанию 2 менее чем в одном случае из 10^{41}. Поэтому, если вы просто ищете большое простое число для каких-то приложений, для всех практических целей почти никогда не возникнет ошибки, если большие числа подбираются случайным образом до тех пор, пока для одного из них процедура PSEUDOPRIME выдаст результат ПРОСТОЕ. Но если числа, которые проверяются на простоту, подбираются не случайным образом, необходим более качественный тест. Как вы увидите, немного сообразительности и рандомизации позволят создать программу проверки простых чисел, которая будет хорошо работать для любых входных данных.

К сожалению, мы не можем устранить все ошибки, просто проверив выполнение (31.40) для другого основания, скажем, для $a = 3$, поскольку существуют составные числа n, удовлетворяющие уравнению (31.40) для *всех* $a \in \mathbb{Z}_n^*$. Эти числа известны как ***числа Кармайкла*** (Carmichael numbers). (Заметим, что уравнение (31.40) не выполняется, когда $\gcd(a, n) > 1$ — т.е. когда $a \notin \mathbb{Z}_n^*$, — но продемонстрировать путем поиска соответствующего a, что n — составное число, может оказаться очень трудно, если n делится только на большие простые числа.) Первые три числа Кармайкла — 561, 1105 и 1729. Числа Кармайкла встречаются крайне редко, например имеется всего 255 таких чисел, меньших 100 000 000. Понять, почему эти числа так редко встречаются, поможет упр. 31.8.2.

В следующем подразделе будет показано, как улучшить тест простоты таким образом, чтобы в нем не возникали ошибки из-за чисел Кармайкла.

Рандомизированный тест простоты Миллера–Рабина

Тест простоты Миллера–Рабина позволяет решить проблемы, возникающие в простом тесте PSEUDOPRIME. Этого удается достичь за счет двух модификаций, описанных ниже.

- В этом тесте выполняется проверка не одного, а нескольких случайным образом выбранных значений a.
- В процессе вычисления каждой степени по модулю в тесте проверяется, не обнаружен ли нетривиальный квадратный корень из 1 по модулю n в ходе последней серии возведений в квадрат. Если это так, то процедура останавливает свою работу и выдает сообщение СОСТАВНОЕ. Правильность выявления составных чисел таким способом подтверждается следствием 31.35 из раздела 31.6.

Ниже приведен псевдокод теста простоты Миллера–Рабина. На его вход подаются проверяемое нечетное число $n > 2$ и значение s — количество случайным образом выбранных значений из множества $\mathbb{Z}_n^+$, относительно которых проверяется простота. В этом коде используется генератор случайных чисел RANDOM, описанный на с. 143: процедура RANDOM$(1, n-1)$ возвращает случайное целое число, удовлетворяющее неравенству $1 \le a \le n-1$. В коде используется вспомогательная процедура WITNESS. Процедура WITNESS(a, n) выдает значение TRUE тогда и только тогда, когда значение a "свидетельствует" о том, что число n составное, т.е. если с помощью a можно доказать (способом, который станет понятен через некоторое время), что n — составное. Проверка WITNESS(a, n) — это более эффективное обобщение теста PSEUDOPRIME, основанного на проверке соотношения

$$a^{n-1} \not\equiv 1 \pmod n$$

при $a = 2$. Сначала представим и обоснуем схему работы процедуры WITNESS, а затем покажем, как она используется в тесте простоты Миллера–Рабина. Пусть $n - 1 = 2^t u$, где $t \ge 1$, а u — нечетное; т.е. бинарное представление значения $n-1$ имеет вид бинарного представления нечетного числа u, после которого следует ровно t нулей. Таким образом, выполняется соотношение $a^{n-1} \equiv (a^u)^{2^t} \pmod n$, поэтому можно вычислить величину $a^{n-1} \bmod n$, сначала вычислив значение $a^u \bmod n$, а затем t раз последовательно возведя результат в квадрат.

```
WITNESS(a, n)
1  Пусть t и u — такие, что t ≥ 1, u нечетно и n − 1 = 2^t u
2  x_0 = MODULAR-EXPONENTIATION(a, u, n)
3  for i = 1 to t
4      x_i = x_{i−1}^2 mod n
5      if x_i == 1 и x_{i−1} ≠ 1 и x_{i−1} ≠ n − 1
6          return TRUE
7  if x_t ≠ 1
8      return TRUE
9  return FALSE
```

В псевдокоде процедуры WITNESS вычисляется величина $a^{n-1} \bmod n$. Для этого сначала в строке 2 вычисляется значение $x_0 = a^u \bmod n$, а затем результат последовательно t раз возводится в квадрат в цикле **for** в строках 3–6.

Применив индукцию по i, можно сделать вывод, что последовательность вычисленных значений $x_0, x_1, \ldots, x_t$ удовлетворяет уравнению $x_i \equiv a^{2^i u} \pmod n$ при $i = 0, 1, \ldots, t$, так что, в частности, $x_t \equiv a^{n-1} \pmod n$. Однако этот цикл может закончиться раньше, если после очередного возведения в квадрат в строке 4 в строках 5 и 6 будет обнаружен нетривиальный квадратный корень из 1. В этом случае работа алгоритма завершается, и он возвращает значение TRUE. Это же значение возвращается и в строках 7 и 8, если значение, вычисленное для $x_t \equiv a^{n-1} \pmod n$, не равно 1. Это именно тот случай, когда процедура PSEUDOPRIME выдает сообщение СОСТАВНОЕ. Если в строке 6 или 8 не возвращается значение TRUE, в строке 9 возвращается значение FALSE.

Теперь докажем, что если процедура WITNESS(a, n) возвращает значение TRUE, то с помощью величины a можно доказать, что число n — составное.

Если процедура WITNESS(a, n) возвращает значение TRUE в строке 8, значит, она обнаружила, что справедливо соотношение $x_t = a^{n-1} \bmod n \neq 1$. Однако если число n — простое, то согласно теореме Ферма (теорема 31.31) для всех $a \in \mathbb{Z}_n^+$ выполняется равенство $a^{n-1} \equiv 1(\pmod n)$. Поэтому n не может быть простым, и неравенство $a^{n-1} \bmod n \neq 1$ доказывает этот факт.

Если процедура WITNESS(a, n) возвращает в строке 6 значение TRUE, значит, она обнаружила, что значение x_{i-1} является нетривиальным квадратным корнем 1 по модулю n, поскольку выполняется соотношение $x_{i-1} \not\equiv \pm 1 \pmod n$, но при этом $x_i \equiv x_{i-1}^2 \equiv 1 \pmod n$. В следствии 31.35 утверждается, что нетривиальный квадратный корень из 1 по модулю n может существовать лишь тогда, когда n — составное. Таким образом, тот факт, что x_{i-1} — нетривиальный квадратный корень из 1 по модулю n, доказывает, что n — составное.

На этом доказательство корректности процедуры WITNESS завершено. Если при вызове процедуры WITNESS(a, n) выдается значение TRUE (в строке 6 или 8), то n — гарантированно составное; доказательство этого факта легко провести, пользуясь значениями a и n.

Сейчас будет представлено краткое альтернативное описание поведения алгоритма WITNESS как функции от последовательности $X = \langle x_0, x_1, \ldots, x_t \rangle$. Это описание окажется полезным впоследствии, в ходе анализа эффективности работы проверки простоты Миллера–Рабина. Заметим, что если при некоторых значениях $0 \leq i < t$ выполняется равенство $x_i = 1$, то остальная часть последовательности в процедуре WITNESS может не вычисляться. В таком случае все значения $x_{i+1}, x_{i+2}, \ldots, x_t$ равны 1, и мы считаем, что в последовательности X на этих позициях находятся единицы. Имеем четыре различных случая.

1. $X = \langle \ldots, d \rangle$, где $d \neq 1$: последовательность X не заканчивается единицей. В строке 8 возвращается значение TRUE; a является свидетельством того, что n — составное (согласно теореме Ферма).
2. $X = \langle 1, 1, \ldots, 1 \rangle$: последовательность X состоит из одних единиц. Возвращается значение FALSE; a не является свидетельством того, что n — составное.
3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: последовательность X заканчивается единицей, и последний элемент последовательности, не равный 1, — элемент -1. Возвращается значение FALSE; a не является свидетельством того, что n — составное.

4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, где $d \neq \pm 1$: последовательность X заканчивается единицей, но последний элемент последовательности, не равный 1, не равен и -1. В строке 6 возвращается значение TRUE; a является свидетельством того, что n — составное, поскольку d является нетривиальным квадратным корнем 1.

Теперь рассмотрим тест Миллера–Рабина на простоту, основанный на применении процедуры WITNESS. Как и раньше, предполагается, что n — нечетное целое число, большее 2.

```
MILLER-RABIN(n, s)
1  for j = 1 to s
2      a = RANDOM(1, n − 1)
3      if WITNESS(a, n)
4          return СОСТАВНОЕ      // Определенно
5  return ПРОСТОЕ                // Почти гарантированно
```

Процедура MILLER-RABIN представляет собой вероятностный поиск доказательства того факта, что число n — составное. В основном цикле (который начинается в строке 1) выбирается s случайных значений величины a из множества $\mathbb{Z}_n^+$ (строка 2). Если одно из этих значений свидетельствует о том, что число n — составное, то процедура MILLER-RABIN в строке 4 выдает значение СОСТАВНОЕ. Такой вывод всегда верный согласно корректности процедуры WITNESS для этого случая. Если в ходе s попыток не было таких свидетельств, то в процедуре MILLER-RABIN предполагается, что нет причин считать число n составным, так что оно считается простым. Скоро можно будет убедиться, что при достаточно бóльших значениях s этот вывод правильный с высокой вероятностью. Тем не менее все же имеется небольшая вероятность, что в процедуре могут неудачно выбираться значения a и что существуют свидетельства того, что n — составное, хотя ни одно из них не было найдено.

Чтобы проиллюстрировать работу процедуры MILLER-RABIN, предположим, что n равно числу Кармайкла 561, так что $n - 1 = 560 = 2^4 \cdot 35$, $t = 4$ и $u = 35$. Если выбрано значение $a = 7$, то из рис. 31.4 в разделе 31.6 видно, что процедура WITNESS вычислит $x_0 \equiv a^{35} \equiv 241 \pmod{561}$, что даст нам последовательность $X = \langle 241, 298, 166, 67, 1 \rangle$. Таким образом, при последнем возведении в квадрат обнаружен нетривиальный корень из 1, поскольку $a^{280} \equiv 67 \pmod{n}$ и $a^{560} \equiv 1 \pmod{n}$. Итак, значение $a = 7$ свидетельствует о том, что число n — составное, процедура WITNESS$(7, n)$ возвращает значение TRUE, а процедура MILLER-RABIN — значение СОСТАВНОЕ.

Если длина числа n равна β бит, то для выполнения процедуры MILLER-RABIN требуется $O(s\beta)$ арифметических операций и $O(s\beta^3)$ битовых операций, поскольку в асимптотическом пределе требуется выполнять не более s возведений в степень по модулю.

Частота ошибок в тесте Миллера–Рабина

Если процедура MILLER-RABIN выводит значение ПРОСТОЕ, то с небольшой вероятностью в ней может быть допущена ошибка. Однако, в отличие от проце-

дуры PSEUDOPRIME, эта вероятность не зависит от n; другими словами, для этой процедуры не существует неблагоприятных входных данных. Зато вероятность ошибки в процедуре MILLER-RABIN зависит от величины s и от того, насколько удачно выбирались значения a. Кроме того, поскольку каждая проверка является более строгой, чем обычная проверка уравнения (31.40), исходя из общих принципов, можно ожидать, что для случайно выбранных целых значений n частота ошибок должна быть очень небольшой. Более точное обоснование представлено в сформулированной ниже теореме.

Теорема 31.38

Если n является нечетным составным числом, то количество свидетельств того, что оно составное, — не менее $(n-1)/2$.

Доказательство. В ходе доказательства теоремы будет показано, что количество значений, не являющихся свидетельствами, не превышает $(n-1)/2$, из чего следует справедливость теоремы.

Начнем с утверждения, что любое значение, которое не является свидетельством, должно быть элементом множества $\mathbb{Z}_n^*$. Почему? Рассмотрим произвольное значение a, не являющееся свидетельством. Оно должно удовлетворять соотношению $a^{n-1} \equiv 1 \pmod n$ или эквивалентному соотношению $a \cdot a^{n-2} \equiv 1 \pmod n$. Таким образом, уравнение $ax \equiv 1 \pmod n$ имеет решение, а именно — a^{n-2}. Согласно следствию 31.21 $\gcd(a,n) \mid 1$, из чего, в свою очередь, следует, что $\gcd(a,n) = 1$. Поэтому a является элементом множества $\mathbb{Z}_n^*$; этому множеству принадлежат все значения оснований, не свидетельствующие о том, что число n — составное.

Чтобы завершить доказательство, покажем, что все значения оснований, не свидетельствующие о том, что число n — составное, не просто содержатся в множестве $\mathbb{Z}_n^*$, но и находятся в истинной подгруппе B группы $\mathbb{Z}_n^*$. (Напомним, что B называется *истинной* подгруппой группы $\mathbb{Z}_n^*$, если она является подгруппой $\mathbb{Z}_n^*$, но не равна $\mathbb{Z}_n^*$.) Согласно следствию 31.16 в этом случае мы имеем $|B| \leq |\mathbb{Z}_n^*|/2$. Поскольку $|\mathbb{Z}_n^*| \leq n-1$, получаем неравенство $|B| \leq (n-1)/2$. Поэтому количество значений оснований, не являющихся свидетельствами, не превышает $(n-1)/2$, следовательно, количество свидетельств должно быть не меньше $(n-1)/2$.

Теперь покажем, как найти истинную подгруппу B группы $\mathbb{Z}_n^*$, которая содержит все значения оснований, не являющиеся свидетельствами. Выделим два случая.

Случай 1. Существует значение $x \in \mathbb{Z}_n^*$, такое, что

$$x^{n-1} \not\equiv 1 \pmod n .$$

Другими словами, n не является числом Кармайкла. Поскольку, как мы уже знаем, числа Кармайкла встречаются крайне редко, случай 1 — основной случай, встречающийся "на практике" (т.е. тогда, когда число n выбрано случайным образом и выполняется проверка его простоты).

Пусть $B = \{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. Ясно, что множество B непустое, так как $1 \in B$. Поскольку группа B замкнута относительно операции умножения по модулю n, из теоремы 31.14 следует, что B — подгруппа группы $\mathbb{Z}_n^*$. Заметим, что каждое значение a, которое не является свидетельством, принадлежит множеству B, поскольку такое a удовлетворяет соотношению $a^{n-1} \equiv 1 \pmod{n}$. Поскольку $x \in \mathbb{Z}_n^* - B$, B является истинной подгруппой группы $\mathbb{Z}_n^*$.

Случай 2. Для всех $x \in \mathbb{Z}_n^*$

$$x^{n-1} \equiv 1 \pmod{n} . \tag{31.41}$$

Другими словами, n — число Кармайкла. На практике этот случай встречается крайне редко. Однако тест MILLER-RABIN (в отличие от теста на псевдопростоту), как сейчас будет показано, в состоянии эффективно определить, что число Кармайкла — составное.

В этом случае число n не может быть степенью простого числа. Чтобы понять, почему это так, предположим обратное, т.е. что $n = p^e$, где p — простое, а $e > 1$. Противоречие мы получим следующим образом. Поскольку предполагается, что n — нечетное, то число p также должно быть нечетным. Из теоремы 31.32 следует, что $\mathbb{Z}_n^*$ — циклическая группа: она содержит генератор g, такой, что $\text{ord}_n(g) = |\mathbb{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. (Формула для $\phi(n)$ получена из (31.20).) Согласно уравнению (31.41) имеем $g^{n-1} \equiv 1 \pmod{n}$. Тогда из теоремы о дискретном логарифме (теорема 31.33 при $y = 0$) следует, что $n - 1 \equiv 0 \pmod{\phi(n)}$, или

$$(p-1)p^{e-1} \mid p^e - 1 .$$

При $e > 1$ мы получаем противоречие, поскольку $(p-1)p^{e-1}$ делится на простое число p, а $p^e - 1$ — не делится. Таким образом, число n не является степенью простого числа.

Поскольку нечетное составное число n не является степенью простого, оно раскладывается на множители $n_1 n_2$, где n_1 и n_2 — взаимно простые нечетные числа, бóльшие единицы. (Таких разложений может быть несколько; в этом случае не играет роли, какое из них выбирается. Например, если $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, то можно выбрать $n_1 = p_1^{e_1}$ и $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Напомним, что значения t и u определены так, что $n - 1 = 2^t u$, где $t \geq 1$, а u — нечетно, и что в процедуре WITNESS для входного значения a вычисляется последовательность

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

(все вычисления проводятся по модулю n).

Назовем пару (v, j) целых чисел ***приемлемой*** (acceptable), если $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, \ldots, t\}$ и

$$v^{2^j u} \equiv -1 \pmod{n} .$$

Приемлемые пары точно существуют, поскольку u нечетно; если выбрать $v = n - 1$ и $j = 0$, то такая пара $(n - 1, 0)$ будет приемлемой. Теперь выберем наибольшее из возможных значений j, для которого существует приемлемая пара

(v, j), и зафиксируем значение v. Пусть

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod n\} .$$

Поскольку множество B замкнуто относительно операции умножения по модулю n, оно является подгруппой группы $\mathbb{Z}_n^*$. Поэтому согласно теореме 31.15 $|B|$ является делителем $|\mathbb{Z}_n^*|$. Каждое значение, которое не является свидетельством, должно быть элементом множества B, поскольку последовательность X, образованная такими значениями, должна либо полностью состоять из единиц, либо содержать -1 не позже чем в j-й позиции в соответствии с условием максимальности значения j. (Если пара (a, j') приемлемая, а значение a не является свидетельством, то из способа выбора значения j должно следовать неравенство $j' \leq j$.)

Теперь воспользуемся фактом существования значения v, чтобы продемонстрировать, что существует элемент $w \in \mathbb{Z}_n^* - B$, а следовательно, что B является истинной подгруппой $\mathbb{Z}_n^*$. Поскольку $v^{2^j u} \equiv -1 \pmod n$, из следствия 31.29 китайской теоремы об остатках вытекает, что $v^{2^j u} \equiv -1 \pmod{n_1}$. Согласно следствию 31.28 существует значение w, которое одновременно удовлетворяет таким уравнениям:

$$\begin{aligned} w &\equiv v \pmod{n_1} , \\ w &\equiv 1 \pmod{n_2} . \end{aligned}$$

Следовательно,

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1} , \\ w^{2^j u} &\equiv 1 \pmod{n_2} . \end{aligned}$$

Согласно следствию 31.29 из соотношения $w^{2^j u} \not\equiv 1 \pmod{n_1}$ следует $w^{2^j u} \not\equiv 1 \pmod n$, а из $w^{2^j u} \not\equiv -1 \pmod{n_2}$ вытекает $w^{2^j u} \not\equiv -1 \pmod n$. Следовательно, можно сделать вывод, что $w^{2^j u} \not\equiv \pm 1 \pmod n$, так что $w \notin B$.

Остается показать, что $w \in \mathbb{Z}_n^*$. Для этого построим рассуждения отдельно по модулю n_1 и по модулю n_2. Что касается операций по модулю n_1, заметим, что, поскольку $v \in \mathbb{Z}_n^*$, справедливо равенство $\gcd(v, n) = 1$, так что и $\gcd(v, n_1) = 1$; если у числа v нет общих делителей с n, то у него также нет общих делителей с n_1. Поскольку $w \equiv v \pmod{n_1}$, можно сделать вывод, что $\gcd(w, n_1) = 1$. Что же касается операций по модулю n_2, заметим, что из соотношения $w \equiv 1 \pmod{n_2}$ следует, что $\gcd(w, n_2) = 1$. Для того, чтобы объединить эти результаты, воспользуемся теоремой 31.6, из которой следует, что $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$, т.е. $w \in \mathbb{Z}_n^*$.

Следовательно, $w \in \mathbb{Z}_n^* - B$, и мы приходим к выводу, что в случае 2 B является истинной подгруппой $\mathbb{Z}_n^*$.

Итак, мы убедились, что в обоих случаях количество свидетельств того, что число n — составное, не меньше $(n-1)/2$. ■

Теорема 31.39

При любом нечетном $n > 2$ и положительном целом s вероятность того, что процедура MILLER-RABIN(n, s) выдаст неправильный результат, не превышает 2^{-s}.

Доказательство. Из теоремы 31.38 следует, что если n — составное, то при каждом выполнении цикла **for** в строках 1–4 вероятность обнаружить свидетельство x того, что n — составное, не меньше $1/2$. Процедура MILLER-RABIN допускает ошибку только в том случае, если ей не удалось обнаружить такое свидетельство в каждой из s итераций основного цикла. Вероятность подобной последовательности неудач не превышает 2^{-s}. ■

Если n — простое, то процедура MILLER-RABIN всегда будет сообщать ПРОСТОЕ, а если n — составное, то вероятность того, что процедура MILLER-RABIN сообщит ПРОСТОЕ, не превышает 2^{-s}.

Однако при применении процедуры MILLER-RABIN к большому случайно выбранному целому числу n необходимо оценить вероятность того, что n — простое, чтобы корректно интерпретировать результаты работы процедуры MILLER-RABIN. Предположим, что мы фиксируем битовую длину β и случайным образом выбираем число n длиной β бит для проверки на простоту. Обозначим через A событие, заключающееся в том, что n — простое число. В соответствии с теоремой о простых числах (теорема 31.37) вероятность того, что n — простое, приближенно равна

$$\begin{aligned}\Pr\{A\} &\approx 1/\ln n \\ &\approx 1.443/\beta\ .\end{aligned}$$

Пусть теперь B означает событие, что процедура MILLER-RABIN вернула ПРОСТОЕ. Мы имеем $\Pr\{\overline{B} \mid A\} = 0$ (или, что эквивалентно, $\Pr\{B \mid A\} = 1$) и $\Pr\{B \mid \overline{A}\} \le 2^{-s}$ (или, что эквивалентно, $\Pr\{\overline{B} \mid \overline{A}\} > 1 - 2^{-s}$).

Но чему равно $\Pr\{A \mid B\}$, вероятность того, что n — простое, если процедура MILLER-RABIN вернула значение ПРОСТОЕ? Согласно альтернативной форме теоремы Байеса (уравнение (В.18)) имеем

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)}\ .\end{aligned}$$

Эта вероятность не превышает $1/2$, пока s не превысит $\lg(\ln n - 1)$. Интуитивно ясно, что большое количество испытаний нужно только для того, чтобы преодолеть недоверие к результату, возникающее из возможной неспособности найти свидетельство того, что число n — составное, при том что составных чисел существенно больше, чем простых. Для чисел длиной $\beta = 1024$ бит это количество

составляет около

$$\lg(\ln n - 1) \approx \lg(\beta/1.443)$$
$$\approx 9$$

испытаний. В любом случае выбор $s = 50$ должен быть достаточен для любого мыслимого приложения.

В действительности ситуация намного лучше. Если пытаться искать большие целые числа, применяя процедуру MILLER-RABIN к большим случайным образом выбранным нечетным целым числам, то выбор небольшого значения s (скажем, 3) приведет к ошибочному результату с очень малой вероятностью, хотя здесь мы и не будем это доказывать. Причина заключается в том, что для случайным образом выбранного составного нечетного целого числа n ожидаемое количество оснований, не являющихся свидетельствами того, что n составное, оказывается гораздо меньшим, чем $(n-1)/2$.

Однако если число n выбирается не случайным образом, то лучшее, что можно доказать с помощью улучшенной версии теоремы 31.38, — это то, что количество значений оснований, не являющихся свидетельствами, не превышает $(n-1)/4$. Более того, существуют такие целые числа n, для которых это количество равно $(n-1)/4$.

Упражнения

31.8.1

Докажите, что если целое нечетное число $n > 1$ не является простым числом или степенью простого числа, то существует нетривиальный квадратный корень из 1 по модулю n.

31.8.2 ★

Теорему Эйлера можно слегка усилить, придав ей следующий вид:

$$a^{\lambda(n)} \equiv 1 \pmod n \text{ для всех } a \in \mathbb{Z}_n^* ,$$

где $n = p_1^{e_1} \cdots p_r^{e_r}$ и $\lambda(n)$ определено как

$$\lambda(n) = \operatorname{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) . \tag{31.42}$$

Докажите, что $\lambda(n) \mid \phi(n)$. Составное число n является числом Кармайкла, если $\lambda(n) \mid n - 1$. Наименьшее из чисел Кармайкла равно $561 = 3 \cdot 11 \cdot 17$; при этом $\lambda(n) = \operatorname{lcm}(2, 10, 16) = 80$, а это делитель 560. Докажите, что числа Кармайкла должны быть “свободными от квадратов” (т.е. не должны делиться на квадрат ни одного простого числа) и в то же время представлять собой произведение не менее трех простых чисел. (По этой причине они встречаются не очень часто.)

31.8.3

Докажите, что если x является нетривиальным квадратным корнем 1 по модулю n, то и $\gcd(x-1, n)$, и $\gcd(x+1, n)$ являются нетривиальными делителями n.

⋆ 31.9. Целочисленное разложение

Предположим, что задано целое число n, которое нужно ***разложить*** (factor) на простые множители. Тест простоты, представленный в предыдущем разделе, может дать информацию о том, что число n — составное, однако он не говорит о его простых множителях. Разложение больших целых чисел n представляется намного более сложной задачей, чем определение того, является ли число n простым или составным. Располагая суперкомпьютерами и наилучшими на сегодняшний день алгоритмами, нереально разложить на множители произвольное 1024-битовое число.

Эвристический ρ-метод Полларда

Пробное деление на каждое целое число вплоть до R гарантирует, что будет полностью разложено любое число вплоть до R^2. Представленная ниже процедура позволяет разложить любое число вплоть до R^4 (если только нам не будет хронически не везти), выполнив тот же объем работы. Поскольку эта процедура носит лишь эвристический характер, ничего нельзя утверждать наверняка ни о времени ее работы, ни о том, что она действительно достигнет успеха. Несмотря на это, данная процедура оказывается весьма эффективной на практике. Другое преимущество процедуры POLLARD-RHO состоит в том, что в ней используется лишь фиксированное количество памяти. (Для небольших чисел ее легко реализовать даже на программируемом калькуляторе.)

```
POLLARD-RHO(n)
 1  i = 1
 2  x_1 = RANDOM(0, n − 1)
 3  y = x_1
 4  k = 2
 5  while TRUE
 6      i = i + 1
 7      x_i = (x_{i−1}^2 − 1) mod n
 8      d = gcd(y − x_i, n)
 9      if d ≠ 1 и d ≠ n
10          print d
11      if i == k
12          y = x_i
13          k = 2k
```

Эта процедура работает следующим образом. Строки 1 и 2 инициализируют i единицей, а x_1 — случайно выбранным из $\mathbb{Z}_n$ значением. Цикл **while**, начинающийся в строке 5, продолжается до бесконечности в поисках множителей числа n. Во время каждой итерации цикла **while** в строке 7 используется рекуррентное соотношение

$$x_i = (x_{i-1}^2 - 1) \bmod n \ , \tag{31.43}$$

позволяющее получить очередное значение x_i бесконечной последовательности

$$x_1, x_2, x_3, x_4, \ldots , \qquad (31.44)$$

а значение индекса i увеличивается в строке 6. Несмотря на то что для простоты восприятия в коде используются значения переменных с индексами x_i, программа работает точно так же, если опустить все индексы, поскольку при каждой итерации необходимо поддерживать только одно последнее значение x_i. С учетом этой модификации в процедуре используется лишь фиксированное количество ячеек памяти.

Время от времени программа сохраняет последние из сгенерированных значений x_i в переменной y. Сохраняются те значения, индексы которых равны степени двойки:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots .$$

В строке 3 сохраняется значение x_1, а в строке 12 сохраняется x_k, когда i становится равным k. Переменная k инициализируется двойкой в строке 4, а строка 13 удваивает ее, когда в строке 12 происходит обновление y. Так что k пробегает значения $1, 2, 4, 8, \ldots$ и всегда дает индекс очередного значения x_k, которое будет сохранено в переменной y.

В строках 8–10 предпринимается попытка разложить число n с помощью сохраненного значения y и текущего значения x_i. В частности, в строке 8 вычисляется наибольший общий делитель $d = \gcd(y - x_i, n)$. Если значение переменной d — нетривиальный делитель числа n (это проверяется в строке 9), то оно выводится в строке 10.

Эта процедура поиска делителей на первый взгляд может показаться несколько загадочной. Однако заметим, что она никогда не выдает неверного ответа; все числа, которые выводит процедура POLLARD-RHO, являются нетривиальными делителями n. Тем не менее эта процедура может вообще не выводить никаких данных; нет никакой гарантии, что она выдаст хоть какой-то результат. Тем не менее, как мы сможем убедиться, есть веская причина ожидать, что процедура POLLARD-RHO выведет множитель p числа n после $\Theta(\sqrt{p})$ итераций цикла **while**. Таким образом, если n — составное число, то эта процедура после приблизительно $n^{1/4}$ обновлений обнаружит достаточное количество делителей, чтобы можно было полностью разложить число n, поскольку все простые множители p числа n, кроме, возможно, наибольшего, меньше $\sqrt{n}$.

Начнем анализ поведения представленной выше процедуры с изучения вопроса о том, сколько времени должно пройти, пока в случайной последовательности по модулю n не повторится значение. Поскольку множество $\mathbb{Z}_n$ конечное и поскольку каждое значение последовательности (31.44) зависит только от предыдущего, то эта последовательность в конце концов начнет повторяться. Достигнув некоторого значения x_i, такого, что при некотором $j < i$ выполняется равенство $x_i = x_j$, последовательность зациклится, поскольку $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$ и т.д. Понять, почему этот метод был назван эвристическим ρ-методом, помогает рис. 31.7. Часть последовательности $x_1, x_2, \ldots, x_{j-1}$ можно изобразить в виде "хвоста" греческой буквы ρ, а цикл $x_j, x_{j+1}, \ldots, x_i$ — в виде "тела" этой буквы.

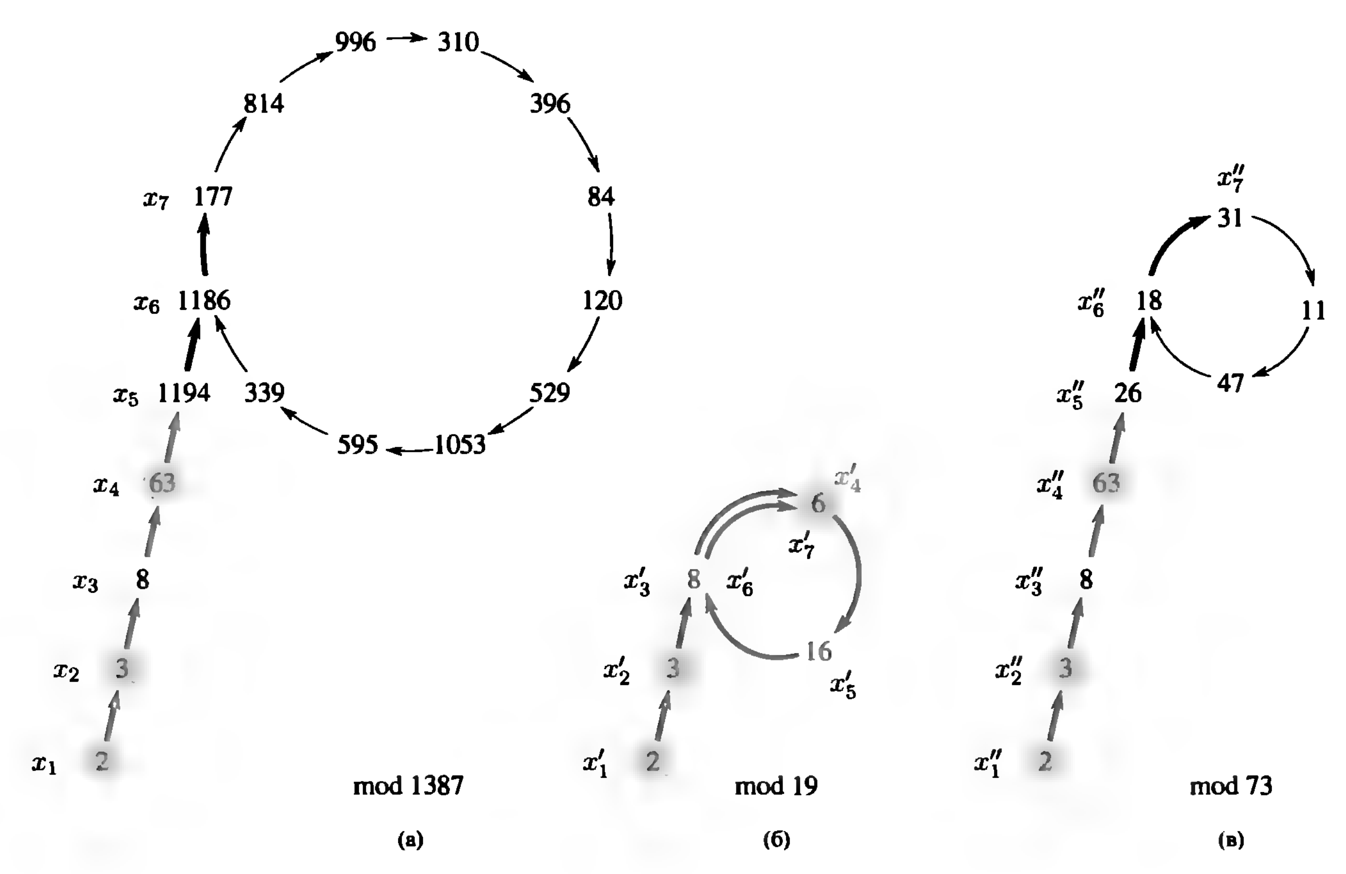


Рис. 31.7. ρ-эвристика Полларда. **(а)** Значения, полученные с помощью рекуррентного соотношения $x_{i+1} = (x_i^2 - 1) \bmod 1387$, начиная с $x_1 = 2$. Разложение числа 1387 на простые множители имеет вид 19·73. Толстые стрелки указывают итерации, выполненные до нахождения множителя 19. Стрелки, изображенные тонкими линиями, указывают на те значения в итерации, которые так и не достигаются. С их помощью иллюстрируется форма буквы ρ. Серым фоном выделены те значения, которые сохраняются процедурой POLLARD-RHO в переменной y. Множитель 19 обнаруживается при достижении значения $x_7 = 177$, когда вычисляется величина $\gcd(63 - 177, 1387) = 19$. Первое значение x, которое впоследствии повторится, равно 1186, однако множитель 19 обнаруживается еще до этого повтора. **(б)** Значения, полученные с помощью того же рекуррентного соотношения, по модулю 19. Каждое значение x_i из части (а) эквивалентно по модулю 19 значению x'_i, показанному в этой части. Например, и $x_4 = 63$, и $x_7 = 177$ эквивалентны 6 по модулю 19. **(в)** Значения, полученные с помощью того же рекуррентного соотношения, по модулю 73. Каждое значение x_i из части (а) эквивалентно по модулю 73 значению x''_i, показанному в этой части. Согласно китайской теореме об остатках каждый узел в части (а) соответствует паре узлов — одному из части (б) и одному из части (в).

Рассмотрим вопрос о том, насколько длинной должна быть последовательность значений x_i, чтобы начать повторяться. Это не совсем то, что нам нужно, но вскоре станет понятно, как модифицировать эти рассуждения. Чтобы оценить это время, будем считать, что функция

$$f_n(x) = (x^2 - 1) \bmod n$$

ведет себя, как "случайная" функция. Конечно, на самом деле она не случайная, но это предположение согласуется с наблюдаемым поведением процедуры POLLARD-RHO. Далее предположим, что каждое значение x_i извлекается независимым образом из множества $\mathbb{Z}_n$ и что все они распределены в этом множестве равномерно. В соответствии с анализом парадокса о днях рождения, проведенном

в разделе 5.4.1, математическое ожидание количества шагов перед зацикливанием последовательности равно $\Theta(\sqrt{n})$.

Теперь перейдем к требуемой модификации. Пусть p — нетривиальный множитель числа n, такой, что $\gcd(p, n/p) = 1$. Например, если разложение числа n имеет вид $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, то в качестве множителя p можно выбрать величину $p_1^{e_1}$. (Полезно запомнить, что если $e_1 = 1$, то роль p будет играть наименьший простой множитель числа n.)

Последовательность $\langle x_i \rangle$ порождает соответствующую последовательность $\langle x_i' \rangle$ по модулю p, где

$$x_i' = x_i \bmod p$$

для всех i.

Кроме того, поскольку в определении функции f_n содержатся только арифметические операции (возведение в квадрат и вычитание) по модулю n, нетрудно показать, что величину x_{i+1}' можно вычислить на основании величины x_i'. Рассмотрение последовательности "по модулю p" — сокращенная версия того, что происходит по модулю n:

$$\begin{aligned}
x_{i+1}' &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2 - 1) \bmod n) \bmod p \\
&= (x_i^2 - 1) \bmod p && \text{(согласно упр. 31.1.7)} \\
&= ((x_i \bmod p)^2 - 1) \bmod p \\
&= ((x_i')^2 - 1) \bmod p \\
&= f_p(x_i') \; .
\end{aligned}$$

Таким образом, хотя мы и не вычислили явным образом последовательность $\langle x_i' \rangle$, она вполне определена и подчиняется тому же рекуррентному соотношению, что и последовательность $\langle x_i \rangle$.

Продолжая рассуждать так же, как и ранее, можно прийти к выводу, что математическое ожидание количества шагов, выполненных перед тем, как повторится последовательность $\langle x_i' \rangle$, равно $\Theta(\sqrt{p})$. Если значение p мало́ по сравнению с n, последовательность $\langle x_i' \rangle$ может начать повторяться намного быстрее, чем последовательность $\langle x_i \rangle$. Действительно, как демонстрируют части (б) и (в) рис. 31.7, последовательность $\langle x_i' \rangle$ начнет повторяться, как только два элемента последовательности $\langle x_i \rangle$ окажутся эквивалентными по модулю p, а не по модулю n.

Обозначим через t индекс первого повторившегося значения последовательности $\langle x_i' \rangle$, а через $u > 0$ — длину цикла, полученного таким образом. Другими словами, t и $u > 0$ — наименьшие значения, для которых при всех $i \geq 0$ выполняется равенство $x_{t+i}' = x_{t+u+i}'$. Согласно приведенным выше рассуждениям, математическое ожидание как величины t, так и u равно $\Theta(\sqrt{p})$. Заметим, что если $x_{t+i}' = x_{t+u+i}'$, то $p \mid (x_{t+u+i} - x_{t+i})$. Таким образом, $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

Поэтому после того как в процедуре POLLARD-RHO в переменной y будет сохранено любое значение x_k, такое, что $k \geq t$, величина $y \bmod p$ всегда будет

находиться в цикле по модулю p. (Если в переменной y будет сохранено новое значение, оно также окажется в цикле по модулю p.) В конце концов переменной k будет присвоено значение, превышающее u, после чего в процедуре будет пройден полный цикл по модулю p, не сопровождающийся изменением значения y. Множитель числа n будет обнаружен тогда, когда x_i "натолкнется" на ранее сохраненное значение y по модулю p, т.е. когда $x_i \equiv y \pmod{p}$.

Скорее всего, найденный множитель будет равен p, хотя случайно это может оказаться число, кратное p. Поскольку математическое ожидание величин t и u равно $\Theta(\sqrt{p})$, математическое ожидание количества шагов, необходимых для получения множителя p, равно $\Theta(\sqrt{p})$.

Есть две причины, по которым этот алгоритм может оказаться не таким быстрым, как ожидается. Во-первых, проведенный выше эвристический анализ времени работы нестрогий, и цикл значений по модулю p может оказаться намного длиннее, чем $\sqrt{p}$. В этом случае алгоритм работает правильно, но намного медленнее, чем хотелось бы. Практически же это представляется сомнительным. Во-вторых, среди делителей числа n, которые выдаются этим алгоритмом, всегда может оказаться один из тривиальных множителей 1 или n. Например, предположим, что $n = pq$, где p и q — простые числа. Может оказаться, что значения t и u для множителя p идентичны значениям t и u для множителя q, и поэтому множитель p будет всегда обнаруживаться в результате выполнения той же операции gcd, в которой обнаруживается множитель q. Поскольку оба множителя находятся одновременно, процедура выдает тривиальный множитель $pq = n$, который бесполезен. На практике и эта проблема кажется несущественной. При необходимости нашу эвристическую процедуру можно перезапустить с другим рекуррентным соотношением вида $x_{i+1} = (x_i^2 - c) \bmod n$. (Значений $c = 0$ и $c = 2$ следует избегать по причинам, вникать в которые мы здесь не станем; но другие значения вполне подходят.)

Конечно же, этот анализ эвристический и нестрогий, поскольку рекуррентное соотношение на самом деле не является "случайным". Тем не менее на практике процедура работает хорошо, и, по-видимому, ее эффективность совпадает с полученной в ходе эвристического анализа. Она представляет собой вероятностный метод поиска небольших простых множителей, на которые раскладывается большое число. Все, что нужно для полного разложения β-битового составного числа n, — найти все его простые множители, меньшие $\lfloor n^{1/2} \rfloor$, поэтому можно ожидать, что процедуре POLLARD-RHO понадобится не более $n^{1/4} = 2^{\beta/4}$ арифметических операций и не более $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ битовых операций. Зачастую наиболее привлекательной особенностью этой процедуры оказывается ее способность находить небольшой множитель p числа n ценой ожидаемого количества $\Theta(\sqrt{p})$ арифметических операций.

Упражнения

31.9.1

Когда процедура POLLARD-RHO выведет множитель 73 числа 1387, если история вычислений в ней имеет вид, показанный на рис. 31.7, (а)?

31.9.2

Предположим, что задана функция $f : \mathbb{Z}_n \to \mathbb{Z}_n$ и начальное значение $x_0 \in \mathbb{Z}_n$. Определим рекуррентное соотношение $x_i = f(x_{i-1})$ для $i = 1, 2, \ldots$. Пусть t и $u > 0$ — наименьшие значения, такие, что $x_{t+i} = x_{t+u+i}$ для $i = 0, 1, \ldots$ (в терминах ρ-алгоритма Полларда t — это длина хвоста, а u — длина цикла ρ). Сформулируйте эффективный алгоритм, позволяющий точно определить значения t и u, и проанализируйте время его работы.

31.9.3

Чему равно математическое ожидание количества шагов, которые понадобится выполнить процедуре POLLARD-RHO, чтобы обнаружить множитель вида p^e, где p — простое число, а $e > 1$?

31.9.4 ★

Как уже упоминалось, одним из недостатков процедуры POLLARD-RHO является то, что на каждом шаге рекурсии в ней необходимо выполнять операцию gcd. Было высказано предположение о том, что вычисления gcd можно выполнить пакетно, накапливая произведение нескольких величин x_i в строке с последующим использованием этого произведения при вычислении gcd вместо величины x_i. Приведите подробное описание того, как можно было бы реализовать эту идею, почему она работает и какой размер пакета можно было бы выбрать для достижения максимального эффекта при обработке β-битового числа n.

Задачи

31.1. Бинарный алгоритм поиска наибольшего общего делителя

На большинстве компьютеров операции вычитания, проверки четности (нечетное или четное данное число) и деления пополам выполняются быстрее, чем вычисление остатка. В этой задаче исследуется ***бинарный алгоритм поиска наибольшего общего делителя*** (binary gcd algorithm), позволяющий избежать вычисления остатков, которые используются в алгоритме Евклида.

а. Докажите, что если и a, и b четны, то $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.

б. Докажите, что если a нечетно, а b четно, то $\gcd(a, b) = \gcd(a, b/2)$.

в. Докажите, что если и a, и b нечетны, то $\gcd(a, b) = \gcd((a - b)/2, b)$.

г. Разработайте эффективный бинарный алгоритм поиска gcd для целых чисел a и b, где $a \geq b$, время работы которого равно $O(\lg a)$. Предполагается, что на каждое вычитание, проверку на четность и деление пополам затрачивается единичное время.

31.2. Анализ битовых операций в алгоритме Евклида

а. Рассмотрим обычный, "карандашом на бумаге", алгоритм деления a на b, в результате выполнения которого получаются частное q и остаток r. Покажите, что в этом методе требуется выполнить $O((1 + \lg q) \lg b)$ битовых операций.

б. Определим функцию $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Покажите, что количество битовых операций, которые выполняются процедурой EUCLID при сведении задачи вычисления величины $\gcd(a, b)$ к вычислению величины $\gcd(b, a \bmod b)$, не превышает $c(\mu(a, b) - \mu(b, a \bmod b))$ для некоторой достаточно большой константы $c > 0$.

в. Покажите, что выполнение алгоритма EUCLID(a, b) требует в общем случае $O(\mu(a, b))$ битовых операций и $O(\beta^2)$ битовых операций, если оба входных значения являются β-битовыми числами.

31.3. Три алгоритма вычисления чисел Фибоначчи

В этой задаче выполняется сравнение производительности работы трех методов вычисления n-го числа Фибоначчи F_n при заданном n. Считаем, что стоимость сложения, вычитания или умножения двух чисел независимо от их размера равна $O(1)$.

а. Покажите, что время работы прямого рекурсивного метода вычисления числа F_n на основе рекуррентного соотношения (3.22) увеличивается экспоненциально с ростом n. (См., например, процедуру FIB на с. 814.)

б. Покажите, как с помощью технологии запоминания вычислить число F_n за время $O(n)$.

в. Покажите, как вычислить число F_n за время $O(\lg n)$, используя только сложения и умножения целых чисел. (*Указание:* рассмотрите матрицу

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

и ее степени.)

г. Предположим теперь, что для сложения двух β-битовых чисел требуется время $\Theta(\beta)$, а для их умножения — время $\Theta(\beta^2)$. Чему равно время работы перечисленных выше алгоритмов с учетом такой стоимости выполнения элементарных арифметических операций?

31.4. Квадратичные вычеты

Пусть p — нечетное целое число. Число $a \in \mathbb{Z}_p^*$ является ***квадратичным вычетом*** (quadratic residue), если уравнение $x^2 = a \pmod p$ имеет решение относительно неизвестного x.

а. Покажите, что существует ровно $(p-1)/2$ квадратичных вычетов по модулю p.

б. Если число p — простое, то определим ***символ Лежандра*** (Legendre symbol) $\left(\frac{a}{p}\right)$ для $a \in \mathbb{Z}_p^*$ как равный 1, если a — квадратичный вычет по модулю p, и -1 в противном случае. Докажите, что если $a \in \mathbb{Z}_p^*$, то

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p .$$

Разработайте эффективный алгоритм, позволяющий определить, является ли заданное число a квадратичным вычетом по модулю p. Проанализируйте время работы этого алгоритма.

в. Докажите, что если p — простое число вида $4k + 3$, а a — квадратичный вычет в $\mathbb{Z}_p^*$, то величина $a^{k+1} \bmod p$ равна квадратному корню a по модулю p. Сколько времени требуется для поиска квадратного корня квадратичного вычета a по модулю p?

г. Разработайте эффективный рандомизированный алгоритм поиска неквадратичного вычета по модулю, равному произвольному простому числу p. Другими словами, нужно найти элемент $\mathbb{Z}_p^*$, который не является квадратичным вычетом. Сколько в среднем операций требуется выполнить этому алгоритму?

Заключительные замечания

В книге Нивена (Niven) и Цукермана (Zuckerman) [263] содержится прекрасное введение в элементарную теорию чисел. У Кнута (Knuth) [209] приведено подробное обсуждение алгоритмов, предназначенных для поиска наибольших общих делителей, а также других основных теоретико-числовых алгоритмов. Бач (Bach) [29] и Ризель (Riesel) [293] представили более современный обзор вычислительных приложений теории чисел. В статье Диксона (Dixon) [90] содержится обзор методов разложения и проверки простоты чисел. В трудах конференции под редакцией Померанца (Pomerance) [278] содержится несколько превосходных обзорных статей. Позже Бач (Bach) и Шаллит (Shallit) [30] представили прекрасный обзор основ вычислительной теории чисел.

В книге Кнута [209][3] обсуждается история возникновения алгоритма Евклида. Он встречается в трактате "*Начала*" греческого математика Евклида, опубликованном в 300 году до н.э., в седьмой книге — в теоремах 1 и 2. Описанный Евклидом алгоритм мог быть получен из алгоритма, предложенного Евдоксом около 375 года до н.э. Не исключено, что алгоритм Евклида является старейшим нетривиальным алгоритмом; соперничать с ним может только алгоритм умноже-

[3]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 2. Получисленные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

ния, известный еще древним египтянам. В статье Шаллита [310] описана история анализа алгоритма Евклида.

Авторство частного случая китайской теоремы об остатках (теоремы 31.27) Кнут приписывает китайскому математику Сунь-Цзы, жившему приблизительно между 200 годом до н.э. и 200 годом н.э. (дата довольно неопределенная). Тот же частный случай сформулирован греческим математиком Никомахусом (Nichomachus) около 100 года н.э. В 1247 году он был обобщен Чином Чиу-Шао (Chhin Chiu-Shao). Наконец в 1734 году Л. Эйлер (L. Euler) сформулировал и доказал китайскую теорему об остатках в общем виде.

Представленный в этой книге рандомизированный алгоритм проверки простоты чисел взят из статей Миллера (Miller) [253] и Рабина (Rabin) [287]; это самый быстрый (с точностью до постоянного множителя) из известных рандомизированных алгоритмов проверки простоты. Доказательство теоремы 31.39 слегка адаптировано по сравнению с тем, что было предложено Бачем (Bach) [28]. Доказательство более строгого результата для процедуры MILLER-RABIN принадлежит Монье (Monier) [256,257].

В течение многих лет проверка чисел на простоту была классическим примером задачи, в которой рандомизация представлялась совершенно необходимой для создания эффективного (с полиномиальным временем работы) алгоритма. Однако в 2002 году Агравал (Agrawal), Каял (Kayal) и Саксема (Saxema) [4] поразили всех открытием детерминированного алгоритма с полиномиальным временем работы для проверки простоты. До того самым быстрым детерминированным алгоритмом был алгоритм Кохена (Cohen) и Ленстры (Lenstra) [72], выполнявшийся для входного числа n за время $(\lg n)^{O(\lg \lg \lg n)}$, что несколько превышает полиномиальное время. Тем не менее для практических целей рандомизированные алгоритмы остаются более эффективными и предпочтительными.

Обстоятельное обсуждение задачи поиска больших "случайных" простых чисел содержится в статье Бьючимина (Beauchemin), Брассарда (Brassard), Крипо (Crépeau), Готье (Goutier) и Померанца (Pomerance) [35].

Концепция криптографической системы с открытым ключом сформулирована Диффи (Diffie) и Хеллманом (Hellman) [86]. Криптографическая система RSA была предложена в 1977 году Ривестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman) [294]. С тех пор в области криптографии были достигнуты большие успехи. Углубилось понимание криптографической системы RSA, и в ее современных реализациях представленные здесь основные методы существенно улучшены. Кроме того, разработаны многие новые методы доказательства безопасности криптографических систем. Например, Голдвассер (Goldwasser) и Микали (Micali) [141] показали, что рандомизация может выступать в роли эффективного инструмента при разработке безопасных схем кодирования с открытым ключом. Голдвассер, Микали и Ривест [142] представили схему цифровой подписи, для которой доказана трудность ее подделки, эквивалентная трудности разложения больших чисел на множители. В книге Менезеса (Menezes), ван Ооршота (van Oorschot) и Ванстоуна (Vanstone) [252] представлен обзор прикладной криптографии.

Эвристический ρ-подход, предназначенный для разложения целых чисел на множители, был изобретен Поллардом (Pollard) [275]. Представленный в этой книге вариант является версией, предложенной Брентом (Brent) [55].

Время работы наилучшего из алгоритмов, предназначенных для разложения больших чисел, приближенно выражается экспоненциальной функцией от кубического корня длины подлежащего разложению числа n. Обобщенный теоретико-числовой алгоритм разложения по принципу решета, разработанный Бахлером (Buhler), Ленстрой (Lenstra) и Померанцем (Pomerance) [56] как обобщение идей, заложенных в теоретико-числовой алгоритм разложения по принципу решета Полларда (Pollard) [276] и Ленстры (Lenstra) и др. [231] и усовершенствованный Копперсмитом (Coppersmith) [76] и другими, по-видимому, наиболее эффективен в общем случае для обработки больших входных чисел. Несмотря на то, что строгий анализ этого алгоритма провести сложно, при разумных предположениях можно оценить время его работы как $L(1/3, n)^{1.902+o(1)}$, где $L(\alpha, n) = e^{(\ln n)^\alpha (\ln\ln n)^{1-\alpha}}$.

Метод эллиптических кривых, предложенный Ленстрой (Lenstra) [232], может оказаться эффективнее для некоторых входных данных, чем теоретико-числовой метод решета, поскольку, как и ρ-метод Полларда, он достаточно быстро позволяет найти небольшой простой множитель p. Время его работы при поиске множителя p оценивается как $L(1/2, p)^{\sqrt{2}+o(1)}$.

Глава 32. Поиск подстрок

В программах, предназначенных для редактирования текста, часто необходимо найти все фрагменты текста, которые совпадают с заданным образцом. Обычно текст — это редактируемый документ, а образец — это искомое слово, введенное пользователем. Эффективные алгоритмы решения этой задачи (часто именуемой сопоставлением строк (string matching)) могут сокращать время реакции текстовых редакторов на действия пользователя. Среди множества приложений алгоритмы поиска подстрок используются, например, для поиска заданных образцов в молекулах ДНК. Поисковые системы в Интернете также используют их при поиске веб-страниц, отвечающих запросам.

Формализуем постановку задачи поиска подстрок (string-matching problem) следующим образом. Предполагается, что текст задан в виде массива $T[1\,..\,n]$ длиной n, а образец (шаблон) — в виде массива $P[1\,..\,m]$ длиной $m \leq n$. Далее, предполагается, что элементы массивов P и T — символы из конечного алфавита Σ. Например, алфавит может иметь вид $\Sigma = \{\texttt{0}, \texttt{1}\}$ или $\Sigma = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$. Символьные массивы P и T часто называют ***строками*** (strings) символов.

Обратимся к рис. 32.1. Мы говорим, что образец P ***встречается в тексте со сдвигом s*** (или, что то же самое, что образец P ***встречается начиная с позиции $s + 1$*** в тексте T), если $0 \leq s \leq n - m$ и $T[s + 1\,..\,s + m] = P[1\,..\,m]$ (т.е. если $T[s + j] = P[j]$ для $1 \leq j \leq m$). Если P встречается в T со сдвигом s, то s называется ***допустимым***, или ***корректным, сдвигом***; в противном случае s является ***недопустимым***, или ***некорректным, сдвигом***. ***Задача поиска подстроки*** представляет собой задачу поиска всех допустимых сдвигов, с которыми заданный образец P встречается в тексте T.

Кроме описанного в разделе 32.1 простейшего алгоритма решения задачи "в лоб", все рассматриваемые в этой главе алгоритмы выполняют определенную

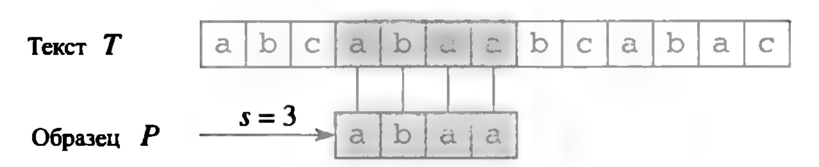


Рис. 32.1. Пример задачи поиска подстроки, в которой необходимо найти все вхождения образца $P = \texttt{abaa}$ в тексте $T = \texttt{abcabaabcabac}$. Образец встречается в тексте только один раз, со сдвигом $s = 3$, который мы называем допустимым сдвигом. Вертикальные линии соединяют каждый символ образца с соответствующим символом текста, и все совпадающие символы заштрихованы.

Алгоритм	Время предварительной обработки	Время сравнения
"В лоб"	0	$O((n-m+1)m)$
Рабина–Карпа	$\Theta(m)$	$O((n-m+1)m)$
Конечный автомат	$O(m\,|\Sigma|)$	$\Theta(n)$
Кнута–Морриса–Пратта	$\Theta(m)$	$\Theta(n)$

Рис. 32.2. Алгоритмы поиска подстрок, представленные в данной главе, и их времена предварительной обработки и сравнения.

предварительную обработку представленного образца, после чего выполняют поиск всех допустимых сдвигов; последняя фаза алгоритма будет называться сравнением, или сопоставлением. На рис. 32.2 приведены времена предварительной обработки и сравнения для каждого из представленных в этой главе алгоритмов. Полное время работы каждого алгоритма равно сумме времени предварительной обработки и времени сравнения. В разделе 32.2 описан интересный алгоритм поиска подстрок, предложенный Рабином (Rabin) и Карпом (Karp). Несмотря на то что время работы этого алгоритма в наихудшем случае (равное $\Theta((n-m+1)m)$ не лучше, чем время работы простейшего метода "в лоб", на практике в среднем он работает намного лучше. Он также легко обобщается на случаи других подобных задач поиска образцов. Затем в разделе 32.3 описывается алгоритм поиска подстрок, работа которого начинается с конструирования конечного автомата, специально предназначенного для поиска совпадений заданного образца P с фрагментами текста. Время предварительной обработки в этом алгоритме равно $O(m\,|\Sigma|)$, зато время сравнения в нем равно всего лишь $\Theta(n)$. Аналогичный, но более совершенный алгоритм Кнута–Морриса–Пратта (Knuth–Morris–Pratt, или KMP) представлен в разделе 32.4; время сравнения в этом алгоритме остается тем же, т.е. оно равно $\Theta(n)$, но время предварительной обработки в нем уменьшается до величины $\Theta(m)$.

Обозначения и терминология

Обозначим через Σ^* множество всех строк конечной длины, образованных с помощью символов алфавита Σ. В этой главе рассматриваются только строки конечной длины. ***Пустая строка*** (empty string), которая обозначается ε и имеет нулевую длину, также принадлежит множеству Σ^*. Длина строки x обозначается $|x|$. ***Конкатенация*** (concatenation) строк x и y, которая обозначается xy, имеет длину $|x| + |y|$ и состоит из символов строки x, после которых следуют символы строки y.

Мы говорим, что строка w является ***префиксом*** (prefix) строки x (обозначается как $w \sqsubset x$), если $x = wy$ для некоторой строки $y \in \Sigma^*$. Заметим, что если $w \sqsubset x$, то $|w| \leq |x|$. Аналогично строку w называют ***суффиксом*** (suffix) строки x (обозначается как $w \sqsupset x$), если существует такая строка $y \in \Sigma^*$, что $x = yw$. Как и в случае префикса, из $w \sqsupset x$ следует неравенство $|w| \leq |x|$. Например, ab $\sqsubset$ abcca и cca $\sqsupset$ abcca. Пустая строка ε является одновременно и суффиксом,

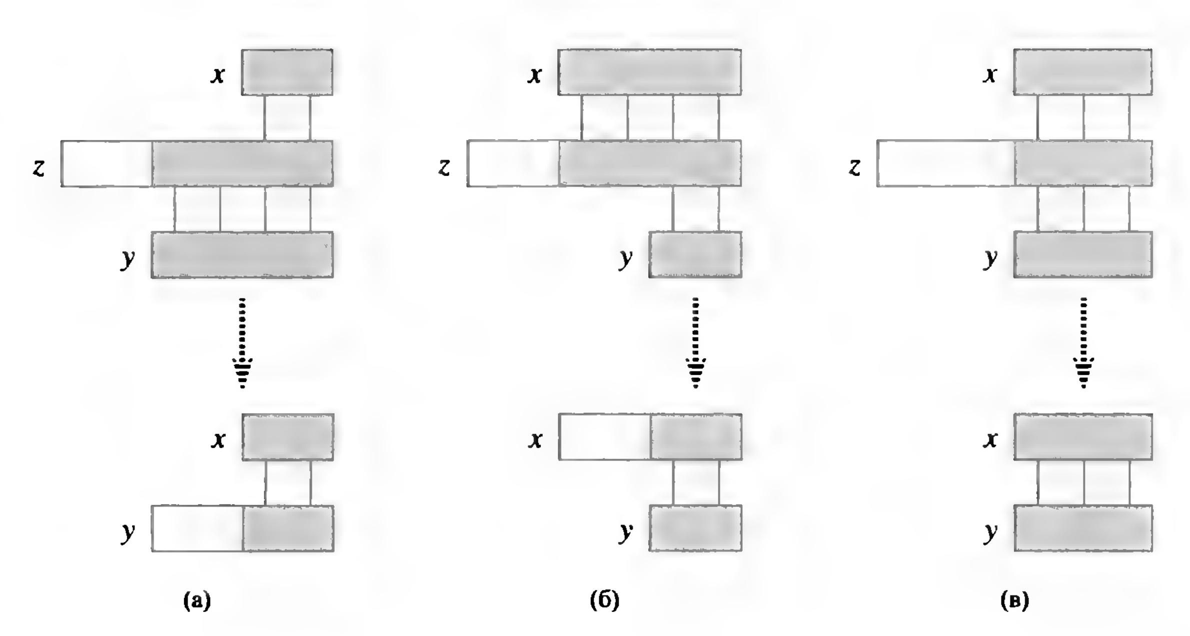


Рис. 32.3. Графическое доказательство леммы 32.1. Предполагаем, что $x \sqsupset z$ и $y \sqsupset z$. На каждой из трех частей рисунка проиллюстрированы три случая леммы. Вертикальными линиями соединены совпадающие области строк (выделенные серым фоном). **(а)** Если $|x| \leq |y|$, то $x \sqsupset y$. **(б)** Если $|x| \geq |y|$, то $y \sqsupset x$. **(в)** Если $|x| = |y|$, то $x = y$.

и префиксом любой строки. Для произвольных строк x и y и для любого символа a соотношение $x \sqsupset y$ выполняется тогда и только тогда, когда $xa \sqsupset ya$. Кроме того, заметим, что отношения $\sqsubset$ и $\sqsupset$ являются транзитивными. Впоследствии окажется полезной сформулированная ниже лемма.

Лемма 32.1 (Лемма о перекрывающихся суффиксах)

Предположим, что x, y и z являются строками, такими, что $x \sqsupset z$ и $y \sqsupset z$. Если $|x| \leq |y|$, то $x \sqsupset y$. Если $|x| \geq |y|$, то $y \sqsupset x$. Если $|x| = |y|$, то $x = y$.

Доказательство. На рис. 32.3 представлено графическое доказательство этой леммы. ■

Обозначим для краткости k-символьный префикс $P[1..k]$ образца $P[1..m]$ как P_k. Таким образом, $P_0 = \varepsilon$ и $P_m = P = P[1..m]$. Аналогично k-символьный префикс текста T обозначим как T_k. С помощью этих обозначений задачу поиска подстрок можно сформулировать как задачу о выявлении всех сдвигов s в интервале $0 \leq s \leq n - m$, таких, что $P \sqsupset T_{s+m}$.

В псевдокоде предполагается, что сравнение двух строк одинаковой длины является примитивной операцией. Если строки сравниваются слева направо и процесс сравнения прерывается, когда обнаружено несовпадение, то считается, что время, которое требуется для подобного теста, выражается линейной функцией от количества совпавших символов. Точнее говоря, считается, что тест "$x == y$" выполняется за время $\Theta(t + 1)$, где t — длина самой длинной строки z, такой, что $z \sqsubset x$ и $z \sqsubset y$. (Чтобы учесть случай, когда $t = 0$, вместо $\Theta(t)$ мы пишем $\Theta(t + 1)$. В этой ситуации не совпадает первый же сравниваемый символ, но для проверки этого требуется некоторое положительное время.)

32.1. Простейший алгоритм поиска подстрок

В простейшем алгоритме поиск всех допустимых сдвигов выполняется с помощью цикла, в котором проверяется условие $P[1 \,..\, m] = T[s+1 \,..\, s+m]$ для каждого из $n - m + 1$ возможных значений s.

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Образец найден со сдвигом" s
```

На рис. 32.4 простейшая процедура поиска подстрок проиллюстрирована как скольжение "шаблона" с образцом по тексту, в процессе которого отмечается, для каких сдвигов все символы шаблона равны соответствующим символам текста. Цикл **for** в строках 3–5 явно исследует каждый сдвиг. Проверка в строке 4 определяет корректность данного сдвига; этот тест неявно включает цикл проверки соответствующих позиций символов до тех пор, пока не выяснится, что все символы совпадают, или пока не будет найдено несоответствие символов. В строке 5 выводится каждый корректный сдвиг s.

Время работы процедуры NAIVE-STRING-MATCHER равно $O((n - m + 1)m)$, и это точная оценка в наихудшем случае. Например, рассмотрим текстовую строку $\mathtt{a}^n$ (строку, состоящую из n символов $\mathtt{a}$) и образец $\mathtt{a}^m$. Для каждого из $n-m+1$ возможных значений сдвига s неявный цикл в строке 4 для проверки совпадения соответствующих символов должен выполнить m итераций, чтобы подтвердить допустимость сдвига. Таким образом, время работы в наихудшем случае равно $\Theta((n - m + 1)m)$, так что в случае $m = \lfloor n/2 \rfloor$ оно становится равным $\Theta(n^2)$. Время работы процедуры NAIVE-STRING-MATCHER равно времени сравнения, так как фаза предварительной обработки отсутствует.

Вскоре вы увидите, что процедура NAIVE-STRING-MATCHER не является оптимальной для этой задачи. В этой главе вы встретитесь с алгоритмом Кнута–Морриса–Пратта, который в наихудшем случае оказывается гораздо лучшим. Про-

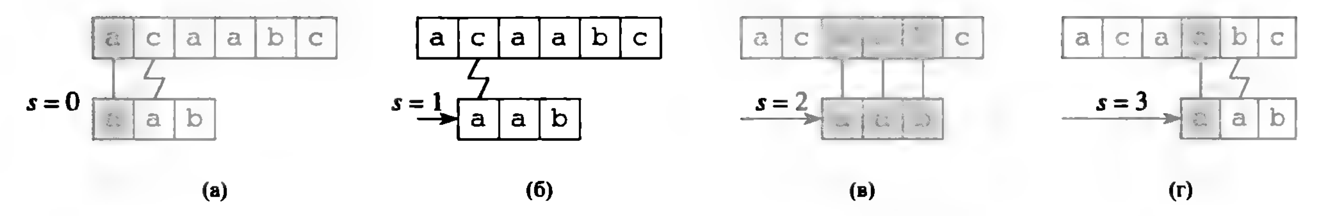


Рис. 32.4. Действия простейшего алгоритма поиска подстрок для образца $P = \mathtt{aab}$ и текста $T = \mathtt{acaabc}$. Можно представить образец P как скользящий вдоль текста. **(а)–(г)** Четыре последовательные размещения, проверенные алгоритмом. В каждой части вертикальные линии соединяют совпадающие области (заштрихованы), а ломаные линии соединяют первые несовпадающие символы, если таковые имеются. Алгоритм обнаруживает в тексте одно совпадение с образцом со сдвигом $s = 2$, показанное в части (в).

стейший алгоритм поиска подстрок оказывается неэффективным, поскольку информация о тексте, полученная для одного значения s, полностью игнорируется при рассмотрении других значений s. Однако эта информация может оказаться очень полезной. Например, если образец имеет вид $P = \mathtt{aaab}$ и если найдено, что сдвиг $s = 0$ допусти́м, то ни один из сдвигов 1, 2 и 3 не может быть допустимым, поскольку $T[4] = \mathtt{b}$. В последующих разделах исследуется несколько способов эффективного использования информации такого рода.

Упражнения

32.1.1

Покажите, какие сравнения выполняются простейшим алгоритмом поиска подстрок, если образец имеет вид $P = \mathtt{0001}$, а текст $T = \mathtt{000010001010001}$.

32.1.2

Предположим, что в образце P все символы различны. Покажите, как ускорить процедуру NAIVE-STRING-MATCHER, чтобы время ее выполнения при обработке n-символьного текста T было равно $O(n)$.

32.1.3

Предположим, что образец P и текст T представляют собой строки длиной m и n соответственно, *случайно* выбранные из d-символьного алфавита $\Sigma_d = \{0, 1, \ldots, d-1\}$, где $d \geq 2$. Покажите, что *математическое ожидание* количества сравнений одного символа с другим, выполняемых в неявном цикле в строке 4 простейшего алгоритма, во всех итерациях этого цикла равно

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1) \ .$$

(Предполагается, что в простейшем алгоритме сравнение символов для данного сдвига прекращается, как только будет обнаружено несовпадение или совпадет весь образец.) Таким образом, для случайных строк простейший алгоритм достаточно эффективен.

32.1.4

Предположим, что в образце P может содержаться специальный ***символ пробела*** (gap character) $\diamond$, который может соответствовать *произвольной* строке (даже нулевой длины). Например, образец $\mathtt{ab}\diamond\mathtt{ba}\diamond\mathtt{c}$ встречается в тексте $\mathtt{cabccbacbacab}$ как

$$\mathtt{c}\underbrace{\mathtt{ab}}_{\mathtt{ab}}\underbrace{\mathtt{cc}}_{\diamond}\underbrace{\mathtt{ba}}_{\mathtt{ba}}\underbrace{\mathtt{cba}}_{\diamond}\underbrace{\mathtt{c}}_{\mathtt{c}}\mathtt{ab}$$

и как

$$\mathtt{c}\underbrace{\mathtt{ab}}_{\mathtt{ab}}\underbrace{\mathtt{ccbac}}_{\diamond}\underbrace{\mathtt{ba}}_{\mathtt{ba}}\underbrace{\quad}_{\diamond}\underbrace{\mathtt{c}}_{\mathtt{c}}\mathtt{ab} \ .$$

Заметим, что в образце символ пробела может встречаться произвольное количество раз, но предполагается, что в тексте он не встречается вообще. Разработайте

алгоритм с полиномиальным временем работы, определяющий, встречается ли заданный образец P в тексте T, и проанализируйте время его работы.

32.2. Алгоритм Рабина–Карпа

Рабин (Rabin) и Карп (Karp) предложили алгоритм поиска подстрок, показывающий на практике хорошую производительность, а также допускающий обобщения на другие родственные задачи, такие как задача о сопоставлении двумерного образца. В алгоритме Рабина–Карпа время $\Theta(m)$ затрачивается на предварительную обработку, а время его работы в наихудшем случае оказывается равным $\Theta((n - m + 1)m)$. Однако с учетом определенных предположений удается показать, что среднее время работы этого алгоритма существенно лучше.

В этом алгоритме используются обозначения из элементарной теории чисел, такие как эквивалентность двух чисел по модулю третьего числа. Соответствующие определения можно найти в разделе 31.1.

Для простоты предположим, что $\Sigma = \{0, 1, 2, \ldots, 9\}$, т.е. что каждый символ представляет собой десятичную цифру. (В общем случае можно предположить, что каждый символ — это цифра в системе счисления с основанием d, где $d = |\Sigma|$.) Теперь строку из k последовательных символов можно рассматривать как десятичное число из k цифр. Таким образом, символьная строка 31415 соответствует числу 31 415. При такой двойной интерпретации входных символов и как графических знаков, и как десятичных цифр в этом разделе удобнее обозначать их цифрами, входящими в стандартный текстовый шрифт.

Для заданного образца $P[1 \mathinner{..} m]$ обозначим через p соответствующее ему десятичное значение. Аналогично для заданного текста $T[1 \mathinner{..} n]$ обозначим через t_s десятичное значение подстрок $T[s+1 \mathinner{..} s+m]$ длиной m при $s = 0, 1, \ldots, n-m$. Очевидно, что $t_s = p$ тогда и только тогда, когда $T[s+1 \mathinner{..} s+m] = P[1 \mathinner{..} m]$; таким образом, s — допустимый сдвиг тогда и только тогда, когда $t_s = p$. Если бы значение p можно было вычислить за время $\Theta(m)$, а все значения t_s — за суммарное время $\Theta(n - m + 1)$[1], то значения всех допустимых сдвигов можно было бы определить за время $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$ путем сравнения значения p с каждым из значений t_s. (Пока что мы не станем беспокоиться по поводу того, что величины p и t_s могут оказаться очень большими числами.)

С помощью правила Горнера (см. раздел 30.1) величину p можно вычислить за время $\Theta(m)$:

$$p = P[m] + 10\left(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots)\right).$$

Аналогично t_0 можно вычислить из $T[1 \mathinner{..} m]$ за то же время $\Theta(m)$.

[1]Мы используем запись $\Theta(n - m + 1)$ вместо $\Theta(n - m)$, поскольку всего имеется $n - m + 1$ различных значений, которые может принимать величина s. Слагаемое "+1" важно в асимптотическом смысле, поскольку при $m = n$ для вычисления единственного значения t_s требуется время $\Theta(1)$, а не $\Theta(0)$.

Чтобы вычислить остальные значения $t_1, t_2, \ldots, t_{n-m}$ за время $\Theta(n-m)$, заметим, что величину t_{s+1} можно вычислить из величины t_s за константное время, так как

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \ . \qquad (32.1)$$

Вычитание $10^{m-1}T[s+1]$ удаляет старшую цифру из t_s, умножение полученного результата на 10 сдвигает число влево на один десятичный разряд, а добавление $T[s+m+1]$ вносит в него соответствующую младшую цифру. Например, если $m = 5$ и $t_s = 31415$, то необходимо удалить старшую цифру $T[s+1] = 3$ и добавить новую младшую цифру (предположим, это $T[s+5+1] = 2$), чтобы получить

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \ . \end{aligned}$$

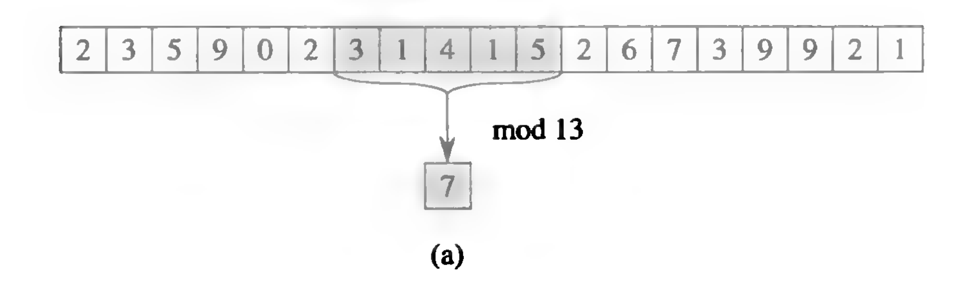
Если предварительно вычислить константу 10^{m-1} (с помощью методов, описанных в разделе 31.6, это можно сделать в течение времени $O(\lg m)$, хотя для данного приложения вполне подойдет обычный метод, требующий времени $O(m)$), то для каждого вычисления результатов выражения (32.1) потребуется фиксированное количество арифметических операций. Таким образом, число p можно вычислить за время $\Theta(m)$, величины $t_0, t_1, \ldots, t_{n-m}$ — за время $\Theta(n-m+1)$, а все вхождения образца $P[1..m]$ в текст $T[1..n]$ можно найти, затратив на фазу предварительной обработки время $\Theta(m)$, а на фазу сравнения — время $\Theta(n-m+1)$.

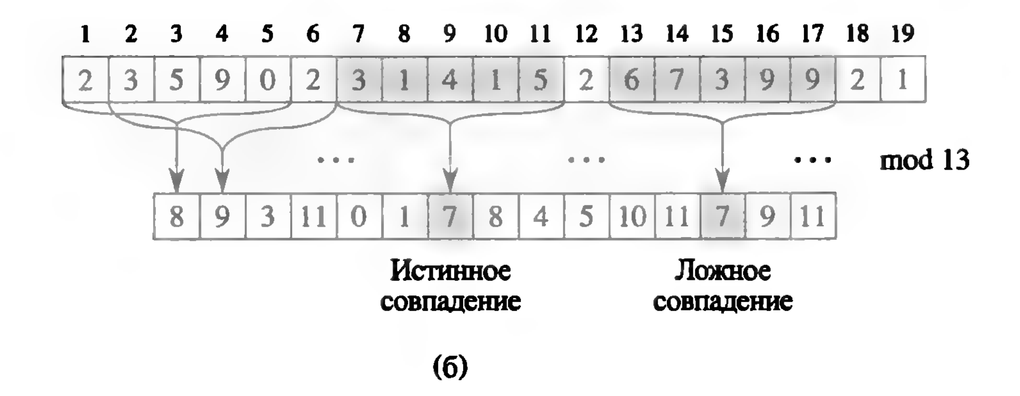
Единственная сложность, возникающая в этой процедуре, может быть связана с тем, что значения p и t_s могут оказаться слишком бóльшими и с ними будет неудобно работать. Если образец P содержит m символов, то предположение о том, что каждая арифметическая операция с числом p (в которое входит m цифр) занимает "фиксированное время", не отвечает действительности. К счастью, эта проблема имеет простое решение (рис. 32.5): вычислять значения p и t_s по модулю некоторого числа q. Значение p по модулю q можно вычислить за время $\Theta(m)$, а все значения t_s по модулю q — за время $\Theta(n-m+1)$. Если выбрать в качестве значения q такое простое число, что $10q$ помещается в компьютерное слово, то все необходимые вычисления можно выполнить с использованием арифметики одинарной точности. В общем случае, если имеется d-символьный алфавит $\{0, 1, \ldots, d-1\}$, значение q выбирается таким образом, чтобы величина dq помещалась в компьютерное слово, и рекуррентное соотношение (32.1) исправляется так, чтобы оно работало по модулю q, т.е. записываем его в виде

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \ , \qquad (32.2)$$

где $h \equiv d^{m-1} \pmod q$ — значение цифры "1" в старшем разряде окна размером m цифр.

Однако решение работать по модулю q не лишено недостатков, поскольку из $t_s \equiv p \pmod q$ не следует, что $t_s = p$. С другой стороны, если $t_s \not\equiv p \pmod q$, то, определенно, выполняется соотношение $t_s \neq p$, так что сдвиг s недопустимый. Таким образом, проверку $t_s \equiv p \pmod q$ можно использовать в качестве





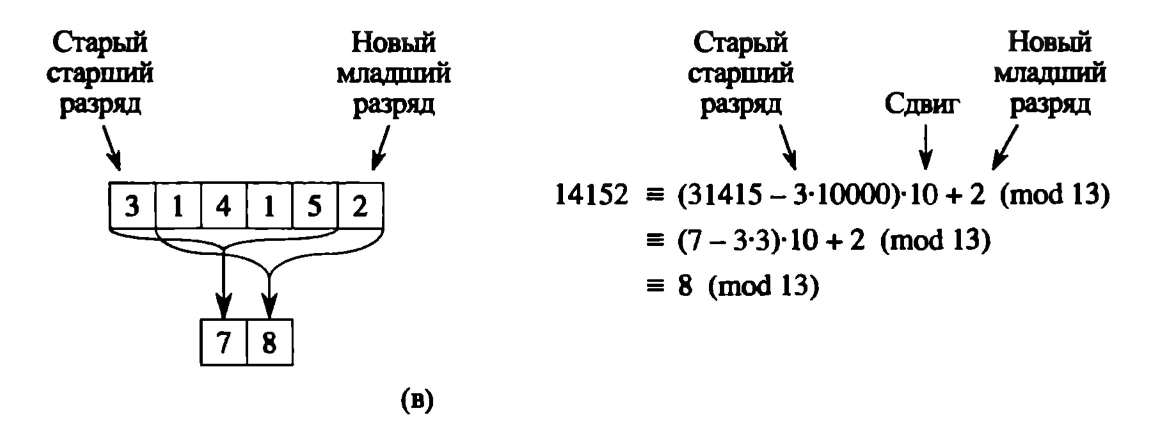


Рис. 32.5. Алгоритм Рабина–Карпа. Каждый символ представляет собой десятичную цифру, а вычисления проводятся по модулю 13. **(а)** Текстовая строка. Окно длиной 5 выделено серым цветом. Выполняется поиск численного значения выделенной подстроки по модулю 13, в результате чего получается значение 7. **(б)** Та же текстовая строка, в которой для всех возможных 5-символьных подстрок вычислены соответствующие им значения по модулю 13. Если предположить, что образец имеет вид $P = 31415$, то нужно найти подстроки, которым соответствуют значения 7 по модулю 13, поскольку $31415 \equiv 7 \pmod{13}$. Алгоритм находит два таких окна, выделенных серой штриховкой. Первое, начинающееся в текстовой позиции 7, действительно представляет собой образец, в то время как второе, начинающееся с позиции 13, является ложным совпадением. **(в)** Как в течение фиксированного времени вычислить значение, соответствующее заданному окну, по значению предыдущего окна. Значение, соответствующее первому окну, равно 31415. Если отбросить цифру 3 в старшем разряде, выполнить сдвиг числа влево (умножить его на 10), а затем добавить к результату цифру 2 в младшем разряде, получится новое значение — 14152. Однако все вычисления выполняются по модулю 13, поэтому для первого окна получается значение 7, а для второго — значение 8.

быстрого эвристического теста, позволяющего исключить недопустимые сдвиги s. Любой сдвиг s, для которого справедливо соотношение $t_s \equiv p \pmod{q}$, необходимо подвергнуть дополнительному тестированию, чтобы проверить, действительно ли этот сдвиг является допустимым или это просто ***ложное совпадение*** (spurious hit). Такое тестирование можно осуществить путем явной проверки условия $P[1\,..\,m] = T[s{+}1\,..\,s{+}m]$. Если значение q достаточно большое, то можно надеяться, что ложные совпадения встречаются довольно редко и стоимость дополнительной проверки окажется низкой.

Приведенная ниже процедура использует описанные выше идеи. В роли входных данных для нее выступают текст T, образец P, основание системы счисления d (в качестве значения которого обычно выбирается $|\Sigma|$) и простое число q.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                        // Предварительная обработка
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m                    // Сравнение
10      if p == t_s
11          if P[1..m] == T[s+1..s+m]
12              print "Образец найден со сдвигом" s
13      if s < n - m
14          t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

Процедура RABIN-KARP-MATCHER работает следующим образом. Все символы рассматриваются как цифры в системе счисления по основанию d. Индексы переменной t приведены для ясности; программа будет правильно работать и без них. В строке 3 переменная h инициализируется цифрой, расположенной в старшем разряде m-цифрового окна. В строках 4–8 вычисляются значение p, равное $P[1\,..\,m] \bmod q$, и значение t_0, равное $T[1\,..\,m] \bmod q$. В цикле **for** в строках 9–14 выполняются итерации по всем возможным сдвигам s. При этом сохраняется сформулированный ниже инвариант.

При выполнении строки 10 справедливо соотношение

$$t_s = T[s+1\,..\,s+m] \bmod q\ .$$

Если в строке 10 выполняется условие $p = t_s$ ("совпадение"), то в строке 11 проверяется справедливость равенства $P[1\,..\,m] = T[s+1\,..\,s+m]$, чтобы исключить ложные совпадения. Все обнаруженные допустимые сдвиги выводятся в строке 12. Если $s < n - m$ (это неравенство проверяется в строке 13), то цикл **for** нужно будет выполнить хотя бы еще один раз, поэтому сначала выполняется

строка 14, чтобы гарантировать соблюдение инварианта цикла, когда мы снова перейдем к строке 10. В строке 14 на основании значения $t_s \bmod q$ с использованием уравнения (32.2) за константное время вычисляется величина $t_{s+1} \bmod q$.

В процедуре RABIN-KARP-MATCHER на предварительную обработку затрачивается время $\Theta(m)$, а время сравнения в нем в наихудшем случае равно $\Theta((n-m+1)m)$, поскольку в алгоритме Рабина–Карпа (как и в простейшем алгоритме поиска подстрок) явно проверяется допустимость каждого сдвига. Если $P = \mathtt{a}^m$, а $T = \mathtt{a}^n$, проверка займет время $\Theta((n-m+1)m)$, поскольку все $n-m+1$ возможных сдвигов являются допустимыми.

Во многих приложениях ожидается небольшое количество допустимых сдвигов (возможно, выражаемое некоторой константой c); в таких приложениях математическое ожидание времени работы алгоритма равно сумме величины $O((n-m+1)+cm) = O(n+m)$ и времени, необходимого для обработки ложных совпадений. В основу эвристического анализа можно положить предположение, что приведение значений по модулю q действует как случайное отображение множества Σ^* на множество $\mathbb{Z}_q$. (См. в разделе 11.3.1 обсуждение вопроса об использовании операции деления для хеширования. Сделанное предположение трудно формализовать и доказать, хотя один из многообещающих подходов заключается в предположении о случайном выборе числа q среди целых чисел подходящего размера. В этой книге такая формализация не применяется.) В таком случае можно ожидать, что число ложных совпадений равно $O(n/q)$, потому что вероятность того, что произвольное число t_s будет эквивалентно p по модулю q, можно оценить как $1/q$. Поскольку имеется всего $O(n)$ позиций, в которых проверка в строке 10 дает отрицательный результат, а на обработку каждого совпадения затрачивается время $O(m)$, математическое ожидание времени сравнения в алгоритме Рабина–Карпа равно

$$O(n) + O(m(v + n/q)) ,$$

где v — количество допустимых сдвигов. Если $v = O(1)$, а q выбрано так, что $q \geq m$, то приведенное выше время выполнения равно $O(n)$. Другими словами, если математическое ожидание количества допустимых сдвигов мало́ ($O(1)$), а выбранное простое число q превышает длину образца, то можно ожидать, что для выполнения фазы сравнения процедуре Рабина–Карпа потребуется время $O(n+m)$. Поскольку $m \leq n$, ожидаемое время сравнения равно $O(n)$.

Упражнения

32.2.1

Сколько ложных совпадений произойдет в процедуре Рабина–Карпа при поиске $P = 26$ в тексте $T = 3141592653589793$, если в качестве модуля q выбрано значение 11?

32.2.2

Как можно обобщить метод Рабина–Карпа для задачи поиска в текстовой строке одного из k заданных образцов? Начните с предположения о том, что все k об-

разцов имеют одинаковую длину. Затем обобщите решение таким образом, чтобы в нем учитывалась возможность того, что образцы могут быть разной длины.

32.2.3

Покажите, как обобщить метод Рабина–Карпа, чтобы он позволял решать задачу поиска заданного образца размером $m \times m$ в символьном массиве размером $n \times n$. (Образец можно сдвигать по вертикали и по горизонтали, но нельзя вращать.)

32.2.4

Алиса располагает копией длинного n-битового файла $A = \langle a_{n-1}, a_{n-2}, \dots, a_0 \rangle$, а Борис — копией n-битового файла $B = \langle b_{n-1}, b_{n-2}, \dots, b_0 \rangle$. Алиса и Борис захотели узнать, идентичны ли их файлы. Чтобы не передавать весь файл A или файл B, они используют описанную ниже быструю вероятностную проверку. Совместными усилиями они выбирают простое число $q > 1000n$, а затем случайным образом выбирают целое число x из множества $\{0, 1, \dots, q-1\}$. После этого Алиса вычисляет значение

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q$$

а Борис — соответствующее значение $B(x)$. Докажите, что если $A \neq B$, то имеется не более одного шанса из 1000, что $A(x) = B(x)$, а если файлы одинаковы, то величины $A(x)$ и $B(x)$ также обязательно совпадут. (*Указание:* см. упр. 31.4.4.)

32.3. Поиск подстрок с помощью конечных автоматов

Многие алгоритмы поиска подстрок начинают выполнение работы с того, что строят конечный автомат — простую машину для обработки информации, — который сканирует строку текста T, выполняя поиск всех вхождений в нее образца P. В этом разделе представлен метод построения такого автомата. Подобные автоматы для поиска подстрок очень эффективны: они проверяют каждый символ текста *ровно по одному разу*, затрачивая на каждый символ фиксированное количество времени. Поэтому после предварительной обработки образца для построения автомата время, необходимое для поиска, равно $\Theta(n)$. Однако время построения автомата может оказаться значительным, если алфавит Σ большой. В разделе 32.4 описан остроумный способ решения этой задачи.

В начале этого раздела дадим определение конечного автомата. Затем мы познакомимся со специальными автоматами, предназначенными для поиска подстрок, и покажем, как с их помощью можно найти все вхождения образца в текст. Наконец будет показано, как сконструировать автомат поиска подстрок для заданного входного образца.

Конечные автоматы

Конечный автомат M, проиллюстрированный на рис. 32.6, представляет собой кортеж из пяти значений $(Q, q_0, A, \Sigma, \delta)$, где

- Q — конечное множество ***состояний***;
- $q_0 \in Q$ — ***начальное состояние***;
- $A \subseteq Q$ — множество различных ***допускающих состояний***;
- Σ — конечный ***входной алфавит***;
- δ — функция, отображающая $Q \times \Sigma$ в Q и называемая ***функцией переходов*** автомата M.

Вначале конечный автомат находится в состоянии q_0 и считывает символы входной строки по одному. Если автомат находится в состоянии q и считывает входной символ a, он совершает переход из состояния q в состояние $\delta(q, a)$. Если текущее состояние q является членом A, то машина M ***принимает***, или ***допускает*** (accepted), строку, считанную к этому моменту. Не принятые входные данные являются ***отвергнутыми*** (rejected).

С конечным автоматом M связана функция ϕ, которая называется ***функцией конечного состояния*** (final state function) и отображает множество Σ^* на множество Q, так, что значение $\phi(w)$ представляет собой состояние, в котором оказывается автомат M после сканирования строки w. Таким образом, M принимает строку w тогда и только тогда, когда $\phi(w) \in A$. Функция ϕ определяется следующим рекуррентным соотношением с использованием функции переходов:

$$\begin{aligned}\phi(\varepsilon) &= q_0 , \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{для } w \in \Sigma^*, a \in \Sigma .\end{aligned}$$

Состояние	Вход a	Вход b
0	1	0
1	0	0

(а)

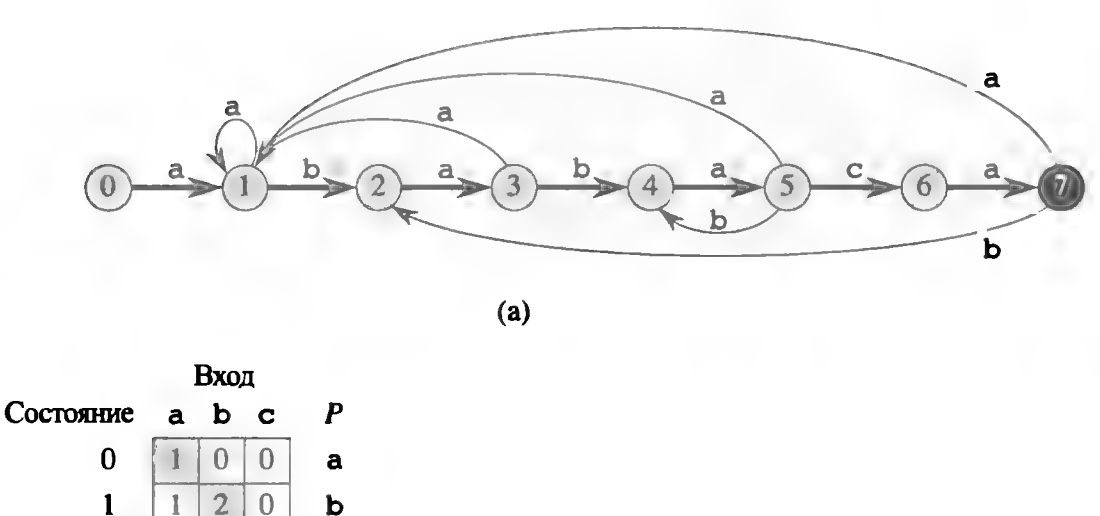
(б)

Рис. 32.6. Простой конечный автомат с двумя состояниями $Q = \{0, 1\}$, начальным состоянием $q_0 = 0$ и входным алфавитом $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. **(а)** Табличное представление функции переходов δ. **(б)** Эквивалентная диаграмма состояний. Состояние 1, показанное черным цветом, — единственное допускающее состояние. Переходы представлены ориентированными ребрами. Например, ребро, ведущее из состояния 1 в состояние 0 и обозначенное меткой b, указывает, что $\delta(1, \mathtt{b}) = 0$. Рассматриваемый автомат принимает те строки, которые оканчиваются нечетным количеством символов a. Точнее говоря, строка x принимается тогда и только тогда, когда $x = yz$, где $y = \varepsilon$ или строка y оканчивается символом b, а $z = \mathtt{a}^k$, где k — нечетное. Например, последовательность состояний, которые проходит этот автомат для входной строки abaaa (включая начальное состояние), имеет вид $\langle 0, 1, 0, 1, 0, 1 \rangle$, так что входная строка принимается. Если входная строка имеет вид abbaa, автомат проходит последовательность состояний $\langle 0, 1, 0, 0, 1, 0 \rangle$, так что входная строка отвергается.

Автоматы поиска подстрок

Для каждого образца P строится свой автомат поиска подстрок; его необходимо сконструировать по образцу на этапе предварительной обработки, чтобы впоследствии использовать для поиска текстовой подстроки. Процесс такого конструирования для образца $P = \texttt{ababaca}$ проиллюстрирован на рис. 32.7. С этого момента предполагается, что образец P — это заданная фиксированная строка: для краткости мы будем опускать в обозначениях зависимость от P.

Чтобы создать автомат поиска подстрок, соответствующий заданному образцу $P[1..m]$, сначала определим вспомогательную функцию σ, которая называется ***суффиксной функцией*** (suffix function), соответствующей образцу P. Функция



(а)

Состояние	Вход a	Вход b	Вход c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(б)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
Состояние $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(в)

Рис. 32.7. (а) Диаграмма состояний автомата поиска подстрок, который воспринимает все строки, оканчивающиеся строкой $\texttt{ababaca}$. Состояние 0 — начальное, а состояние 7 (оно выделено черным цветом) — единственное допускающее. Функция переходов δ определяется уравнением (32.4), а ориентированные ребра, направленные из состояния i в состояние j и обозначенные меткой a, представляют значение функции $\delta(i, a) = j$. Ребра, направленные вправо, которые образуют "хребет" автомата и выделены на рисунке жирными линиями, соответствуют точным совпадениям образца и входных символов. За исключением ребер из состояния 7 в состояния 1 и 2, ребра, направленные влево, соответствуют несовпадениям. Некоторые ребра, соответствующие несовпадениям, не показаны; в соответствии с соглашением, если для некоторого символа $a \in \Sigma$ у состояния i отсутствуют исходящие ребра с меткой a, то $\delta(i, a) = 0$. **(б)** Соответствующая функция переходов δ, а также строка образца $P = \texttt{ababaca}$. Элементы, соответствующие совпадениям между образцом и входными символами, выделены серым цветом. **(в)** Обработка автоматом текста $T = \texttt{abababacaba}$. Под каждым символом текста $T[i]$ указано состояние $\phi(T_i)$, в котором находится автомат после обработки префикса T_i. Образец встречается в тексте один раз, причем совпадающая подстрока оканчивается в девятой позиции.

σ является отображением множества Σ^* на множество $\{0, 1, \ldots, m\}$, таким, что величина $\sigma(x)$ равна длине максимального префикса P, который является суффиксом строки x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} \ . \tag{32.3}$$

Суффиксная функция σ вполне определена, поскольку пустая строка $P_0 = \varepsilon$ является суффиксом любой строки. В качестве примера рассмотрим образец $P = \texttt{ab}$; тогда $\sigma(\varepsilon) = 0$, $\sigma(\texttt{ccaca}) = 1$ и $\sigma(\texttt{ccab}) = 2$. Для образца P длиной m равенство $\sigma(x) = m$ выполняется тогда и только тогда, когда $P \sqsupset x$. Из определения суффиксной функции следует, что если $x \sqsupset y$, то $\sigma(x) \leq \sigma(y)$.

Определим автомат поиска подстрок, соответствующий образцу $P[1..m]$, следующим образом.

- Множество состояний Q имеет вид $\{0, 1, \ldots, m\}$. Начальным состоянием q_0 является состояние 0, а состояние m является единственным принимающим состоянием.
- Функция переходов δ определена для любого состояния q и символа a уравнением

$$\delta(q, a) = \sigma(P_q a) \ . \tag{32.4}$$

Мы определяем $\delta(q, a) = \sigma(P_q a)$, потому что хотим отследить самый длинный префикс образца P, который до сих пор соответствовал текстовой строке T. Чтобы подстрока T — скажем, подстрока, заканчивающаяся в $T[i]$ — соответствовала некоторому префиксу P_j образца P, этот префикс P_j должен быть суффиксом T_i. Предположим, что $q = \phi(T_i)$, так что после чтения T_i автомат находится в состоянии q. Разработаем функцию переходов δ таким образом, чтобы этот номер состояния, q, указывал длину наибольшего префикса P, который соответствует суффиксу T_i, т.е. чтобы в состоянии q выполнялись соотношения $P_q \sqsupset T_i$ и $q = \sigma(T_i)$. (Когда $q = m$, все m символов P соответствуют суффиксу T_i, так что мы находим искомую подстроку.) Таким образом, поскольку и $\phi(T_i)$, и $\sigma(T_i)$ равны q, мы увидим (ниже, в теореме 32.4), что автомат поддерживает следующий инвариант:

$$\phi(T_i) = \sigma(T_i) \ . \tag{32.5}$$

Если автомат находится в состоянии q и считывает очередной символ $T[i+1] = a$, то необходимо, чтобы переход вел в состояние, соответствующее наибольшему префиксу P, который одновременно является суффиксом $T_i a$, а этим состоянием является $\sigma(T_i a)$. Поскольку P_q представляет собой наибольший префикс P, являющийся суффиксом T_i, наибольший префикс P, который является суффиксом $T_i a$, не только $\sigma(T_i a)$, но и $\sigma(P_q a)$. (Лемма 32.3 на с. 1046 доказывает, что $\sigma(T_i a) = \sigma(P_q a)$.) Таким образом, необходимо, чтобы, когда автомат находится в состоянии q, функция переходов для символа a переводила автомат в состояние $\sigma(P_q a)$.

Следует рассмотреть два случая. В первом случае $a = P[q + 1]$, так что символ a продолжает соответствие образцу; при этом, поскольку $\delta(q, a) = q + 1$, переход продолжает выполняться вдоль "хребта" автомата (полужирные ребра на рис. 32.7). Во втором случае $a \neq P[q+1]$, так что a не соответствует образцу. Здесь

нужно найти меньший префикс P, который одновременно является суффиксом T_i. Поскольку на этапе предварительной обработки при создании автомата образец сопоставляется с самим собой, функция переходов быстро находит наибольший среди таких меньших префиксов P.

Давайте обратимся к конкретному примеру. Автомат на рис. 32.7 имеет $\delta(5, \mathtt{c}) = 6$, что иллюстрирует первый случай, когда продолжается соответствие. Чтобы проиллюстрировать второй случай, заметим, что автомат на рис. 32.7 имеет $\delta(5, \mathtt{b}) = 4$. Мы выполняем такой переход, поскольку, если автомат считывает b в состоянии $q = 5$, то $P_q\mathtt{b} = \mathtt{ababab}$, и наибольшим префиксом P, одновременно являющимся суффиксом ababab, является $P_4 = \mathtt{abab}$.

Чтобы пояснить работу автомата, предназначенного для поиска подстрок, приведем простую, но эффективную программу для моделирования поведения такого автомата (представленного функцией переходов δ) при поиске образца P длиной m во входном тексте $T[1\,..\,n]$. Как и в любом другом автомате поиска подстрок, предназначенном для образца длиной m, множество состояний Q имеет вид $\{0, 1, \ldots, m\}$, начальным состоянием является состояние 0, а единственным принимающим является состояние m.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n = T.length
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Образец найден со сдвигом" i − m
```

Из простой циклической структуры процедуры FINITE-AUTOMATON-MATCHER можно легко увидеть, что время поиска совпадений с его помощью в тексте длиной n равно $\Theta(n)$. Однако сюда не входит время предварительной обработки, которое необходимо для вычисления функции переходов δ. К этой задаче мы обратимся позже, после доказательства корректности работы процедуры FINITE-AUTOMATON-MATCHER.

Рассмотрим, как автомат обрабатывает входной текст $T[1\,..\,n]$. Докажем, что после сканирования символа $T[i]$ автомат окажется в состоянии $\sigma(T_i)$. Поскольку соотношение $\sigma(T_i) = m$ справедливо тогда и только тогда, когда $P \sqsupset T_i$, машина окажется в принимающем состоянии m тогда и только тогда, когда она только что считает образец P. Чтобы доказать этот результат, воспользуемся двумя приведенными ниже леммами, в которых идет речь о суффиксной функции σ.

Лемма 32.2 (Неравенство суффиксной функции)

Для любой строки x и символа a выполняется неравенство $\sigma(xa) \leq \sigma(x) + 1$.

Доказательство. Введем обозначение $r = \sigma(xa)$ (рис. 32.8). Если $r = 0$, то неравенство $\sigma(xa) = r \leq \sigma(x) + 1$ тривиальным образом следует из того, что величина $\sigma(x)$ неотрицательна. Теперь предположим, что $r > 0$. Тогда $P_r \sqsupset xa$ согласно определению функции σ. Если в конце строк P_r и xa отбросить по символу

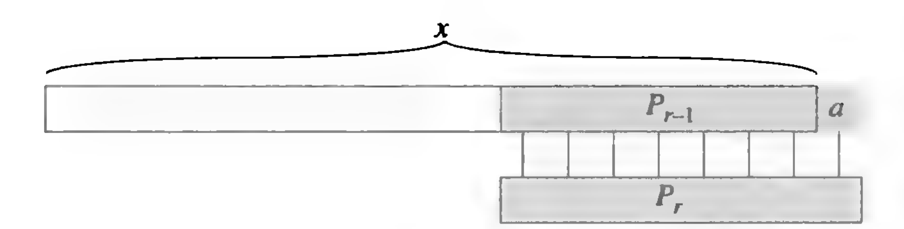


Рис. 32.8. Иллюстрация к доказательству леммы 32.2. На рисунке показано, что $r \leq \sigma(x) + 1$, где $r = \sigma(xa)$.

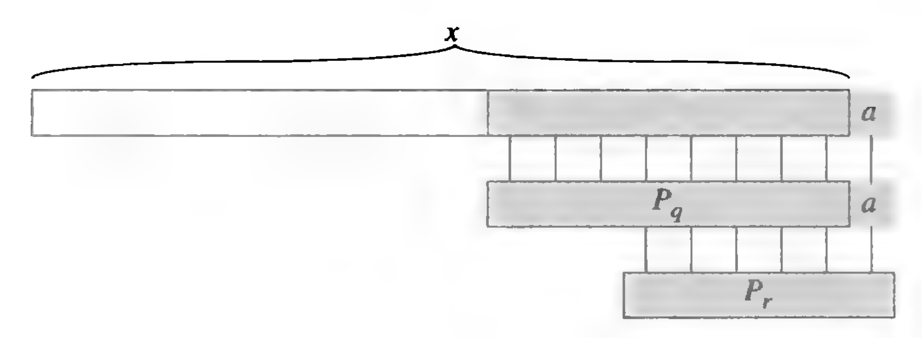


Рис. 32.9. Иллюстрация к доказательству леммы 32.3. На рисунке показано, что $r = \sigma(P_q a)$, где $q = \sigma(x)$ и $r = \sigma(xa)$.

a, то получим $P_{r-1} \sqsupset x$. Следовательно, справедливо неравенство $r - 1 \leq \sigma(x)$, так как $\sigma(x)$ — наибольшее значение k, при котором выполняется соотношение $P_k \sqsupset x$, и, таким образом, $\sigma(xa) = r \leq \sigma(x) + 1$. ∎

Лемма 32.3 (Лемма о рекурсии суффиксной функции)

Если для произвольной строки x и символа a выполняется $q = \sigma(x)$, то $\sigma(xa) = \sigma(P_q a)$.

Доказательство. $P_q \sqsupset x$ из определения σ. Как показано на рис. 32.9, выполняется также $P_q a \sqsupset xa$. Если обозначить $r = \sigma(xa)$, то $P_r \sqsupset xa$ и, согласно лемме 32.2, $r \leq q + 1$. Таким образом, имеем $|P_r| = r \leq q + 1 = |P_q a|$. Поскольку $P_q a \sqsupset xa$, $P_r \sqsupset xa$ и $|P_r| \leq |P_q a|$, из леммы 32.1 следует, что $P_r \sqsupset P_q a$. Следовательно, $r \leq \sigma(P_q a)$, т.е. $\sigma(xa) \leq \sigma(P_q a)$. Но, кроме того, $\sigma(P_q a) \leq \sigma(xa)$, поскольку $P_q a \sqsupset xa$. Таким образом, $\sigma(xa) = \sigma(P_q a)$. ∎

Теперь мы готовы доказать основную теорему, характеризующую поведение автомата поиска подстрок при обработке входного текста. Как уже упоминалось, в этой теореме показано, что автомат на каждом шагу просто отслеживает, какой самый длинный префикс образца совпадает с суффиксом части текста, считанной до сих пор. Другими словами, в автомате поддерживается инвариант (32.5).

Теорема 32.4

Если ϕ — функция конечного состояния автомата поиска подстрок для заданного образца P, а $T[1..n]$ — входной текст автомата, то

$$\phi(T_i) = \sigma(T_i)$$

Доказательство. Доказательство выполняется по индукции по i. Для $i = 0$ теорема тривиально истинна, поскольку $T_0 = \varepsilon$. Таким образом, $\phi(T_0) = 0 = \sigma(T_0)$.

Теперь предположим, что $\phi(T_i) = \sigma(T_i)$ и докажем, что $\phi(T_{i+1}) = \sigma(T_{i+1})$. Обозначим $\phi(T_i)$ как q, а $T[i+1]$ — как a. Тогда

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(согласно определению } T_{i+1} \text{ и } a\text{)} \\
&= \delta(\phi(T_i), a) && \text{(согласно определению } \phi\text{)} \\
&= \delta(q, a) && \text{(согласно определению } q\text{)} \\
&= \sigma(P_q a) && \text{(согласно определению } \delta \text{ (32.4))} \\
&= \sigma(T_i a) && \text{(согласно лемме 32.3 и гипотезе индукции)} \\
&= \sigma(T_{i+1}) && \text{(согласно определению } T_{i+1}\text{)} .
\end{aligned}$$

■

В соответствии с теоремой 32.4, если автомат попадает в состояние q в строке 4, то q — наибольшее значение, такое, что $P_q \sqsupset T_i$. Таким образом, в строке 5 равенство $q = m$ выполняется тогда и только тогда, когда только что был считан образец P. Итак, мы приходим к выводу, что процедура FINITE-AUTOMATON-MATCHER работает корректно.

Вычисление функции переходов

Приведенная ниже процедура вычисляет функцию переходов δ по заданному образцу $P[1..m]$.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for каждого символа a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k − 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

В этой процедуре функция $\delta(q, a)$ вычисляется непосредственно, исходя из ее определения (32.4). Во вложенных циклах, которые начинаются в строках 2 и 3, рассматриваются все состояния q и все символы a, а в строках 4–8 функции $\delta(q, a)$ присваивается наибольшее значение k, такое, что $P_k \sqsupset P_q a$. Код начинается с наибольшего мыслимого значения k, которое равно $\min(m, q+1)$. Затем он уменьшает k до тех пор, пока не будет достигнуто $P_k \sqsupset P_q a$, что в конечном итоге произойдет, поскольку $P_0 = \varepsilon$ является суффиксом любой строки.

Время работы процедуры COMPUTE-TRANSITION-FUNCTION равно $O(m^3 |\Sigma|)$, поскольку внешние циклы дают вклад, соответствующий умножению на $m |\Sigma|$, внутренний цикл **repeat** может повториться не более $m + 1$ раз, а для выполне-

ния проверки $P_k \sqsupset P_q a$ в строке 7 может потребоваться сравнить вплоть до m символов. Существуют процедуры, работающие намного быстрее; время, необходимое для вычисления функции δ по заданному образцу P, путем применения некоторых величин, остроумно вычисленных для этого образца (см. упр. 32.4.8), можно улучшить до величины $O(m\,|\Sigma|)$. С помощью такой усовершенствованной процедуры вычисления функции δ можно найти все вхождения образца длиной m в текст длиной n для алфавита Σ, затратив на предварительную обработку время $O(m\,|\Sigma|)$, а на сравнение — время $\Theta(n)$.

Упражнения

32.3.1

Сконструируйте автомат поиска подстрок для образца P = aabab и проиллюстрируйте его работу при обработке текста T = aaababaabaababaab.

32.3.2

Изобразите диаграмму состояний для автомата поиска подстрок, если образец имеет вид ababbabbababbababbabb, а алфавит — $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

32.3.3

Образец P называется ***строкой с уникальными префиксами*** (nonoverlappable), если из соотношения $P_k \sqsupset P_q$ следует, что $k = 0$ или $k = q$. Как выглядит диаграмма переходов автомата для поиска подстрок с уникальными префиксами?

32.3.4 ★

Заданы два образца — P и P'. Опишите процесс построения конечного автомата, позволяющего найти все вхождения *любого из* образцов. Попытайтесь свести к минимуму количество состояний такого автомата.

32.3.5

Задан образец P, содержащий пробельные символы (см. упр. 32.1.4). Опишите процесс построения конечного автомата, позволяющего найти все вхождения образца P в текст T, затратив при этом на сравнение время $O(n)$, где $n = |T|$.

★ 32.4. Алгоритм Кнута–Морриса–Пратта

Здесь вашему вниманию представим алгоритм сравнения строк, который был предложен Кнутом (Knuth), Моррисом (Morris) и Праттом (Pratt). В нем удается избежать вычисления функции переходов δ, а благодаря использованию вспомогательной функции π, которая вычисляется по заданному образцу за время $\Theta(m)$ и хранится в массиве $\pi[1..m]$, время сравнения в этом алгоритме оказывается равным $\Theta(n)$. Массив π позволяет эффективно (в амортизированном смысле) вычислять функцию δ "на лету", т.е. по мере необходимости. Грубо говоря, для любого состояния $q = 0, 1, \ldots, m$ и любого символа $a \in \Sigma$ величина $\pi[q]$ содер-

жит информацию, необходимую для вычисления величины $\delta(q, a)$, но не зависит от a. Поскольку массив π содержит только m элементов (в то время как в массиве δ их $\Theta(m\,|\Sigma|)$, вычисляя на этапе предварительной обработки функцию π вместо функции δ, удается уменьшить время предварительной обработки образца в $|\Sigma|$ раз.

Префиксная функция для образца

Префиксная функция π для некоторого образца инкапсулирует сведения о том, в какой мере образец совпадает сам с собой после сдвигов. Эта информация позволяет избежать ненужных проверок в простейшем алгоритме поиска подстрок и предвычисления функции δ при использовании конечных автоматов.

Рассмотрим работу обычного алгоритма непосредственного сопоставления. На рис. 32.10, (а) показан некоторый сдвиг s шаблона с образцом P = ababaca вдоль текста T. В этом примере $q = 5$ символов успешно совпали (они выделены серым цветом и соединены вертикальными линиями), а шестой символ образца отличается от соответствующего символа текста. Информация о количестве сов-

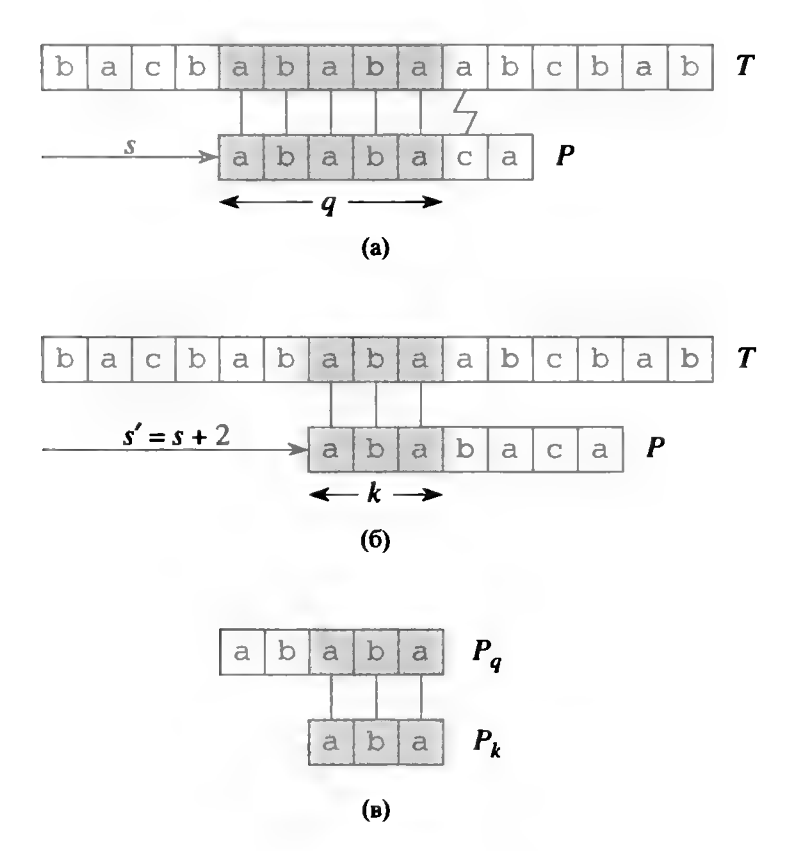


Рис. 32.10. Префиксная функция π. **(а)** Образец P = ababaca, сдвинутый относительно T так, что совпадают первые $q = 5$ символов. Совпадающие символы выделены серой штриховкой и соединены вертикальными линиями. **(б)** Пользуясь одной лишь информацией о пяти совпадающих символах, можно вывести, что сдвиг $s + 1$ недопустим, но сдвиг $s' = s + 2$ согласуется со всем, что мы знаем о тексте, а значит, потенциально допустим. **(в)** Можно предварительно вычислить полезную информацию для таких выводов, сравнив образец с самим собой. Здесь мы видим, что наидлиннейший префикс P, который одновременно является истинным суффиксом P_5, есть P_3. Представим эту предвычисленную информацию в массиве π, так что $\pi[5] = 3$. Если известно, что при сдвиге s наблюдается соответствие q символов, то следующий потенциально допустимый сдвиг, как показано в части (б), — $s' = s + (q - \pi[q])$.

павших символов q позволяет сделать вывод о том, какие символы содержатся в соответствующих местах текста. Эти знания позволяют сразу же определить, что некоторые сдвиги будут недопустимыми. В представленном на рисунке примере сдвиг $s + 1$ точно недопустим, поскольку первый символ образца (a) находился бы напротив символа текста, для которого известно, что он совпадает со вторым символом образца (b). Однако из части (б) рисунка, на которой показан сдвиг $s' = s + 2$, видно, что первые три символа образца совпадают с тремя последними просмотренными символами текста, так что такой сдвиг априори нельзя отбросить как недопустимый. В общем случае полезно знать ответ на следующий вопрос.

> Известно, что символы $P[1..q]$ образца соответствуют символам $T[s + 1..s+q]$ текста. Каков наименьший сдвиг $s' > s$, такой, что для некоторого $k < q$
> $$P[1..k] = T[s' + 1..s' + k] , \qquad (32.6)$$
> где $s' + k = s + q$?

Другими словами, зная, что $P_q \sqsupset T_{s+q}$, необходимо найти наидлиннейший истинный префикс P_k строки P_q, который одновременно является суффиксом T_{s+q}. (Поскольку $s' + k = s + q$, если заданы s и q, то поиск наименьшего сдвига s' эквивалентен поиску длины наибольшего префикса k.) Мы добавляем разность $q - k$ длин этих префиксов P к сдвигу s, чтобы получить новый сдвиг s', так что $s' = s + (q - k)$. В лучшем случае $k = 0$, так что $s' = s + q$ и мы тут же пропускаем сдвиги $s+1, s+2, \ldots, s+q-1$. В любом случае при новом сдвиге s' не нужно сравнивать первые k символов P с соответствующими символами T, поскольку (32.6) гарантирует, что они совпадают.

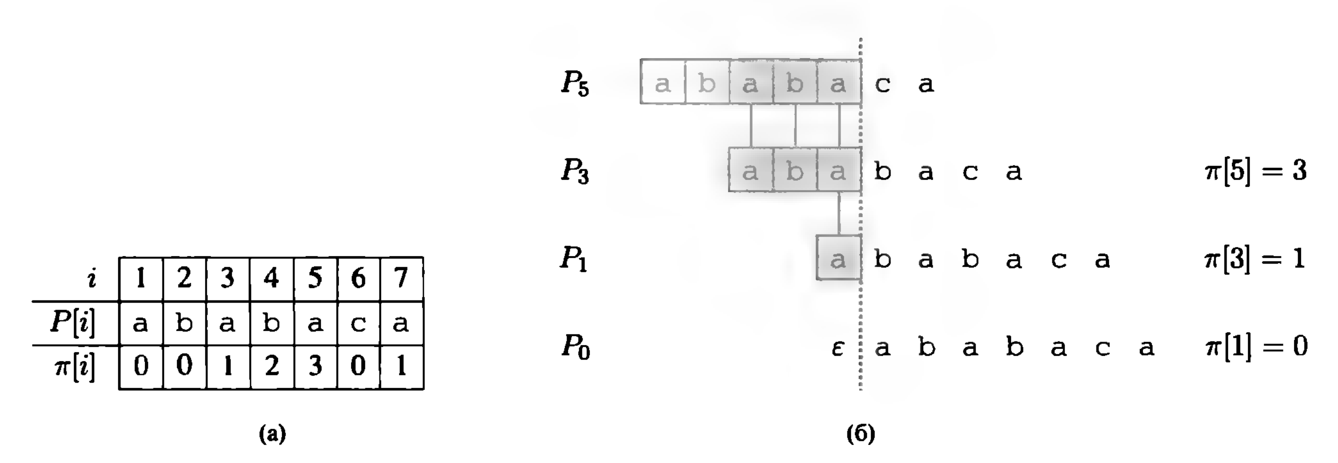
Необходимую информацию можно вычислить путем сравнения образца с самим собой, как показано на рис. 32.10, (в). Поскольку $T[s' + 1..s' + k]$ является частью известного фрагмента текста, она является суффиксом строки P_q. Поэтому уравнение (32.6) можно рассматривать как запрос о максимальном значении $k < q$, таком, что $P_k \sqsupset P_q$. Тогда следующий потенциально допустимый сдвиг равен $s' = s + (q - k)$. Оказывается, что удобнее хранить для каждого значения q количество k совпадающих при новом сдвиге s' символов, чем, например, величину $s' - s$.

Формализуем предварительно вычисляемую информацию следующим образом. Для заданного образца $P[1..m]$ ***префиксной функцией*** для P является функция $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m - 1\}$, такая, что

$$\pi[q] = \max\{k : k < q \text{ и } P_k \sqsupset P_q\} .$$

Иначе говоря, $\pi[q]$ равно длине наибольшего префикса образца P, который является истинным суффиксом строки P_q. В качестве примера на рис. 32.11, (а) приведена полная префиксная функция π для образца ababaca.

Алгоритм поиска подстрок Кнута–Морриса–Пратта приведен ниже в виде псевдокода процедуры KMP-MATCHER. В основном, как вы увидите, эта процедура вытекает из процедуры FINITE-AUTOMATON-MATCHER. В процедуре KMP-



i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Рис. 32.11. Иллюстрация леммы 32.5 для образца $P =$ ababaca и $q = 5$. **(а)** Функция π для указанного образца. Поскольку $\pi[5] = 3$, $\pi[3] = 1$ и $\pi[1] = 0$, итерируя π, получаем $\pi^*[5] = \{3, 1, 0\}$. **(б)** Мы смещаем образец P вправо и замечаем, когда некоторый префикс P_k образца P соответствует некоторому истинному суффиксу P_5; мы получаем соответствие при $k = 3$, 1 и 0. На рисунке в первой строке приведен образец P, а пунктирная вертикальная линия проведена после P_5. Последовательные строки показывают все сдвиги P, которые приводят к тому, что некоторый префикс P_k строки P соответствует некоторому суффиксу P_5. Совпадающие символы выделены штриховкой и соединены вертикальными линиями. Таким образом, $\{k : k < 5 \text{ и } P_k \sqsupset P_5\} = \{3, 1, 0\}$. Лемма 32.5 утверждает, что $\pi^*[q] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$ для всех q.

MATCHER вызывается вспомогательная процедура COMPUTE-PREFIX-FUNCTION, в которой вычисляется функция π.

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                          // Количество совпадающих символов
for i = 1 to n                 // Сканирование текста слева направо
    while q > 0 и P[q + 1] ≠ T[i]
        q = π[q]               // Следующий символ не совпадает
    if P[q + 1] == T[i]
        q = q + 1              // Следующий символ совпадает
    if q == m                  // Совпадает ли весь образец P?
        print "Образец находится со смещением" i − m
        q = π[q]               // Ищем следующее совпадение
```

```
COMPUTE-PREFIX-FUNCTION(P)
1  m = P.length
2  Пусть π[1..m] — новый массив
3  π[1] = 0
4  k = 0
5  for q = 2 to m
6      while k > 0 и P[k + 1] ≠ P[q]
7          k = π[k]
8      if P[k + 1] == P[q]
9          k = k + 1
10     π[q] = k
11 return π
```

Эти две процедуры имеют много общего, поскольку обе сравнивают строки с образцом P: KMP-MATCHER сопоставляет текст T с образцом P, и COMPUTE-PREFIX-FUNCTION — образец P с самим собой.

Начнем с анализа времени работы этой процедуры. Доказательство ее корректности окажется более сложным.

Анализ времени работы

С помощью методов группового анализа (см. раздел 17.1) можно показать, что время работы процедуры COMPUTE-PREFIX-FUNCTION равно $\Theta(m)$. Единственной тонкостью при этом является показ того, что всего цикл **while** в строках 6 и 7 выполняется $O(m)$ раз. Покажем, что он делает не более $m - 1$ итерации. Начнем с некоторых наблюдений над величиной k. Во-первых, в строке 4 значение k равно 0, а единственный путь увеличения k — операция инкремента в строке 9, выполняемая не более одного раза за итерацию цикла **for** в строках 5–10. Таким образом, общее увеличение k составляет не более $m - 1$. Во-вторых, поскольку $k < q$ до входа в цикл **for** и каждая итерация цикла увеличивает q, всегда выполняется $k < q$. Следовательно, присваивание в строках 3 и 10 гарантирует, что $\pi[q] < q$ для всех $q = 1, 2, \ldots, m$, что означает, что каждая итерация цикла **while** уменьшает k. В-третьих, k никогда не становится отрицательным. Объединив эти факты, мы видим, что общее уменьшение k в цикле **while** ограничено сверху общим увеличением k во всех итерациях цикла **for**, которое составляет $m-1$. Таким образом, общее количество итераций цикла **while** не превышает $m - 1$, и время работы процедуры COMPUTE-PREFIX-FUNCTION составляет $\Theta(m)$.

В упр. 32.4.4 предлагается показать с помощью аналогичного группового анализа, что время сравнения процедуры KMP-MATCHER равно $\Theta(n)$.

Благодаря использованию функции π вместо функции δ, которая используется в процедуре FINITE-AUTOMATON-MATCHER, время предварительной обработки образца уменьшается с $O(m\,|\Sigma|)$ до $\Theta(m)$, при том что время фактического сравнения остается ограниченным $\Theta(n)$.

Корректность вычисления префиксной функции

Немного позже мы покажем, что префиксная функция π помогает имитировать функцию переходов δ в автомате поиска подстрок. Но сначала докажем, что процедура COMPUTE-PREFIX-FUNCTION действительно корректно вычисляет эту функцию. Для этого найдем все префиксы P_k, являющиеся истинными суффиксами данного префикса P_q. Значение $\pi[q]$ дает наибольший такой префикс, но следующая лемма, проиллюстрированная на рис. 32.11, показывает, что, итерируя префиксную функцию π, фактически можно перечислить все префиксы P_k, являющиеся истинными суффиксами P_q. Положим

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\} ,$$

где $\pi^{(i)}[q]$ определена в терминах функциональной итерации, так что $\pi^{(0)}[q] = q$ и $\pi^{(i)}[q] = \pi[\pi^{(i-1)}[q]]$ для $i \geq 1$, и где последовательность $\pi^*[q]$ останавливается по достижении $\pi^{(t)}[q] = 0$.

Лемма 32.5 (Лемма об итерации префиксной функции)

Пусть P — образец длиной m с префиксной функцией π. Тогда для $q = 1, 2, \ldots, m$ имеем $\pi^*[q] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$.

Доказательство. Сначала докажем, что $\pi^*[q] \subseteq \{k : k < q \text{ и } P_k \sqsupset P_q\}$ или, что эквивалентно,

$$\text{из } i \in \pi^*[q] \text{ следует } P_i \sqsupset P_q \; . \tag{32.7}$$

Если $i \in \pi^*[q]$, то $i = \pi^{(u)}[q]$ для некоторого $u > 0$. Докажем уравнение (32.7) по индукции по u. Для $u = 1$ имеем $i = \pi[q]$, и наше утверждение следует из того, что $i < q$ и $P_{\pi[q]} \sqsupset P_q$ по определению π. Используя отношения $\pi[i] < i$ и $P_{\pi[i]} \sqsupset P_i$ и транзитивность $<$ и $\sqsupset$, устанавливаем справедливость утверждения для всех i из $\pi^*[q]$. Следовательно, $\pi^*[q] \subseteq \{k : k < q \text{ и } P_k \sqsupset P_q\}$.

Теперь докажем, что $\{k : k < q \text{ и } P_k \sqsupset P_q\} \subseteq \pi^*[q]$, методом от противного. Предположим, что множество $\{k : k < q \text{ и } P_k \sqsupset P_q\} - \pi^*[q]$ непустое, и пусть j представляет собой наибольшее число этого множества. Поскольку $\pi[q]$ является наибольшим значением в $\{k : k < q \text{ и } P_k \sqsupset P_q\}$ и $\pi[q] \in \pi^*[q]$, должно выполняться $j < \pi[q]$, так что пусть j' обозначает наименьшее целое из $\pi^*[q]$, большее, чем j. (Можно выбрать $j' = \pi[q]$, если никакие другие числа в $\pi^*[q]$ не превышают j.) Имеем $P_j \sqsupset P_q$, поскольку $j \in \{k : k < q \text{ и } P_k \sqsupset P_q\}$, и из $j' \in \pi^*[q]$ и уравнения (32.7) получаем $P_{j'} \sqsupset P_q$. Таким образом, $P_j \sqsupset P_{j'}$ согласно лемме 32.1, и j является наибольшим значением, меньшим j' и обладающим этим свойством. Следовательно, должны выполняться $\pi[j'] = j$ и, поскольку $j' \in \pi^*[q]$, $j \in \pi^*[q]$. Это противоречие и доказывает лемму. ■

Алгоритм COMPUTE-PREFIX-FUNCTION вычисляет $\pi[q]$ по порядку для $q = 1, 2, \ldots, m$. Присвоение $\pi[1]$ значение 0 в строке 3 процедуры COMPUTE-PREFIX-FUNCTION, определенно, корректно, поскольку $\pi[q] < q$ для всех q. Воспользуемся для доказательства того, что процедура COMPUTE-PREFIX-FUNCTION корректно вычисляет $\pi[q]$ для $q > 1$, следующей леммой и ее следствием.

Лемма 32.6

Пусть P является образцом длиной m и пусть π представляет собой префиксную функцию для P. Если для $q = 1, 2, \ldots, m$ выполняется $\pi[q] > 0$, то $\pi[q] - 1 \in \pi^*[q-1]$.

Доказательство. Пусть $r = \pi[q] > 0$, так что $r < q$ и $P_r \sqsupset P_q$; таким образом, $r-1 < q-1$ и $P_{r-1} \sqsupset P_{q-1}$ (отбрасывая последние символы из P_r и P_q, что можно сделать, поскольку $r > 0$). Таким образом, согласно лемме 32.5 $r - 1 \in \pi^*[q-1]$. А значит, $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. ■

Определим для $q = 2, 3, \ldots, m$ подмножество $E_{q-1} \subseteq \pi^*[q-1]$ как

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ и } P_k \sqsupset P_{q-1} \text{ и } P[k+1] = P[q]\} \qquad \text{(из леммы 32.5)} \\ &= \{k : k < q-1 \text{ и } P_{k+1} \sqsupset P_q\} \ . \end{aligned}$$

Множество E_{q-1} состоит из значений $k < q - 1$, для которых $P_k \sqsupset P_{q-1}$ и для которых имеем $P_{k+1} \sqsupset P_q$, поскольку $P[k+1] = P[q]$. Таким образом, E_{q-1} состоит из значений $k \in \pi^*[q-1]$, таких, что можно расширить P_k до P_{k+1} и получить истинный суффикс P_q.

Следствие 32.7

Пусть P является образцом длиной m и пусть π представляет собой префиксную функцию для P. Для $q = 2, 3, \ldots, m$

$$\pi[q] = \begin{cases} 0 \ , & \text{если } E_{q-1} = \emptyset \ , \\ 1 + \max\{k \in E_{q-1}\} \ , & \text{если } E_{q-1} \neq \emptyset \ . \end{cases}$$

Доказательство. Если E_{q-1} — пустое множество, не существует $k \in \pi^*[q-1]$ (включая $k = 0$), для которого можно расширить P_k до P_{k+1} и получить истинный префикс P_q. Следовательно, $\pi[q] = 0$.

Если множество E_{q-1} непустое, то для каждого $k \in E_{q-1}$ имеем $k + 1 < q$ и $P_{k+1} \sqsupset P_q$. Следовательно, из определения $\pi[q]$ имеем

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\} \ . \qquad (32.8)$$

Заметим, что $\pi[q] > 0$. Пусть $r = \pi[q] - 1$, так что $r + 1 = \pi[q]$ и, следовательно, $P_{r+1} \sqsupset P_q$. Поскольку $r + 1 > 0$, имеем $P[r+1] = P[q]$. Кроме того, согласно лемме 32.6 имеем $r \in \pi^*[q-1]$. Следовательно, $r \in E_{q-1}$, и, таким образом, $r \leq \max\{k \in E_{q-1}\}$, или, что эквивалентно,

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\} \ . \qquad (32.9)$$

Объединение уравнений (32.8) и (32.9) завершает доказательство. ■

Теперь завершим доказательство того, что процедура COMPUTE-PREFIX-FUNCTION корректно вычисляет функцию π. В этой процедуре в начале каждой итерации цикла **for** в строках 5–10 выполняется равенство $k = \pi[q-1]$. Это условие обеспечивается строками 3 и 4 во время первого вхождения в цикл и остается справедливым в каждой последующей итерации благодаря строке 10. В строках 6–9 значение переменной k изменяется таким образом, что становится равным корректному значению $\pi[q]$. В цикле **while**, в строках 6 и 7, выполняется поиск по всем значениям $k \in \pi^*[q-1]$, пока не будет найдено такое значение k, для которого $P[k+1] = P[q]$. В этот момент k является наибольшим значением множества E_{q-1}, так что согласно следствию 32.7 величину $\pi[q]$ можно положить равной $k+1$. Если же искомое значение $k \in \pi^*[q-1]$, такое, что $P[k+1] = P[q]$, в цикле **while** не найдено, в строке 8 k равно нулю. Если $P[1] = P[q]$, то необходимо присвоить значение 1 как k, так и $\pi[q]$; в противном случае переменную k следует оставить неизменной, а величине $\pi[q]$ присвоить значение 0. В любом случае в строках 8–10 переменные k и $\pi[q]$ получают правильные значения. На этом доказательство корректности процедуры COMPUTE-PREFIX-FUNCTION можно считать завершенным.

Корректность алгоритма Кнута–Морриса–Пратта

Процедуру KMP-MATCHER можно рассматривать как видоизмененную реализацию процедуры FINITE-AUTOMATON-MATCHER, использующую префиксную функцию π для вычисления переходов между состояниями. В частности, докажем, что на i-й итерации циклов **for** как процедуры KMP-MATCHER, так и процедуры FINITE-AUTOMATON-MATCHER состояние q имеет то же значение, что и при тестировании на равенство m (в строке 10 процедуры KMP-MATCHER и в строке 5 процедуры FINITE-AUTOMATON-MATCHER). Поскольку ранее было показано, что процедура KMP-MATCHER имитирует поведение процедуры FINITE-AUTOMATON-MATCHER, корректность процедуры KMP-MATCHER следует из корректности процедуры FINITE-AUTOMATON-MATCHER (и вскоре станет понятно, почему в процедуре KMP-MATCHER необходима строка 12).

Прежде чем формально доказать, что процедура KMP-MATCHER корректно имитирует процедуру FINITE-AUTOMATON-MATCHER, давайте разберемся, как префиксная функция π замещает функцию переходов δ. Вспомним, что когда автомат поиска подстрок находится в состоянии q и сканирует символ $a = T[i]$, он переходит в новое состояние $\delta(q, a)$. Если $a = P[q+1]$, так что a продолжает соответствовать образцу, то $\delta(q, a) = q+1$. В противном случае $a \neq P[q+1]$, так что a не соответствует образцу, и $0 \leq \delta(q, a) \leq q$. В первом случае, когда a продолжает соответствовать, процедура KMP-MATCHER переходит в состояние $q+1$, не обращаясь к функции π: проверка в цикле **while** в строке 6 дает значение FALSE, проверка в строке 8 дает TRUE и строка 9 увеличивает q.

Функция π вступает в игру, когда символ a не соответствует образцу, так что новое состояние $\delta(q, a)$ либо представляет собой q, либо находится слева от q вдоль "хребта" автомата. Цикл **while** в строках 6 и 7 процедуры KMP-MATCHER итеративно проходит по состояниям в $\pi^*[q]$, останавливаясь, либо когда входит

в состояние, скажем, q', такое, что a соответствует $P[q'+1]$, либо когда q' проходит весь путь вниз до нуля. Если a соответствует $P[q'+1]$, то строка 9 устанавливает новое состояние равным $q'+1$, что должно быть равно $\delta(q,a)$ для корректной работы имитации. Другими словами, новое состояние $\delta(q,a)$ должно либо представляет собой состояние 0, либо быть на единицу больше некоторого состояния в $\pi^*[q]$.

Давайте взглянем на пример на рис. 32.7 и 32.11, где образец представляет собой $P = \texttt{ababaca}$. Предположим, что автомат находится в состоянии $q = 5$; состояниями в $\pi^*[5]$ в убывающем порядке являются 3, 1 и 0. Если следующий сканируемый символ — c, то, как легко видеть, автомат переходит в состояние $\delta(5, \texttt{c}) = 6$ как в процедуре FINITE-AUTOMATON-MATCHER, так и в процедуре KMP-MATCHER. Теперь предположим, что вместо этого очередным сканируемым символом является b, так что автомат должен перейти в состояние $\delta(5, \texttt{b}) = 4$. Выход из цикла **while** в процедуре KMP-MATCHER происходит после однократного выполнения строки 7, и автомат оказывается в состоянии $q' = \pi[5] = 3$. Поскольку $P[q'+1] = P[4] = \texttt{b}$, проверка в строке 8 возвращает TRUE, и процедура KMP-MATCHER переходит в новое состояние $q'+1 = 4 = \delta(5, \texttt{b})$. Наконец предположим, что следующий сканируемый символ — a, так что автомат должен перейти в состояние $\delta(5, \texttt{a}) = 1$. При первых трех выполнениях теста в строке 6 возвращается TRUE. В первый раз мы находим, что $P[6] = \texttt{c} \neq \texttt{a}$, и процедура KMP-MATCHER переходит в состояние $\pi[5] = 3$ (первое состояние в $\pi^*[5]$). Во второй раз мы обнаруживаем, что $P[4] = \texttt{b} \neq \texttt{a}$, и переходим в состояние $\pi[3] = 1$ (второе состояние в $\pi^*[5]$). В третий раз мы находим, что $P[2] = \texttt{b} \neq \texttt{a}$, и перемещаемся в состояние $\pi[1] = 0$ (последнее состояние в $\pi^*[5]$). Выход из цикла **while** осуществляется по достижении состояния $q' = 0$. Теперь в строке 8 обнаруживается, что $P[q'+1] = P[1] = \texttt{a}$, и в строке 9 автомат переходит в новое состояние $q'+1 = 1 = \delta(5, \texttt{a})$.

Таким образом, наше интуитивное представление заключается в том, что процедура KMP-MATCHER выполняет итерации по состояниям в $\pi^*[q]$ в убывающем порядке, останавливаясь в некотором состоянии q', а затем, возможно, переходя в состояние $q'+1$. Хотя может показаться, что требуется большое количество работы просто для имитации вычисления $\delta(q,a)$, помните, что асимптотически процедура KMP-MATCHER ничуть не медленнее процедуры FINITE-AUTOMATON-MATCHER.

Теперь мы готовы формально доказать корректность алгоритма Кнута–Морриса–Пратта. Согласно теореме 32.4 после каждого выполнения строки 4 процедуры FINITE-AUTOMATON-MATCHER мы имеем $q = \sigma(T_i)$. Следовательно, достаточно показать, что то же самое свойство выполняется и в случае цикла **for** в процедуре KMP-MATCHER. Доказательство выполняется по индукции по числу итераций цикла. Изначально обе процедуры устанавливают q равным нулю при первом входе в соответствующие циклы **for**. Рассмотрим итерацию i цикла **for** процедуры KMP-MATCHER, и пусть q' — состояние в начале данной итерации цикла. Согласно гипотезе индукции $q' = \sigma(T_{i-1})$. Необходимо показать, что $q = \sigma(T_i)$ в строке 10. (Строка 12 будет обработана отдельно.)

Когда мы рассматриваем символ $T[i]$, наидлиннейшим префиксом P, являющимся суффиксом T_i, является либо $P_{q'+1}$ (если $P[q'+1] = T[i]$), либо некоторый префикс (необязательно истинный и, возможно, пустой) $P_{q'}$. Рассмотрим по отдельности три случая, когда $\sigma(T_i) = 0$, $\sigma(T_i) = q' + 1$ и $0 < \sigma(T_i) \leq q'$.

- Если $\sigma(T_i) = 0$, то $P_0 = \varepsilon$ является единственным префиксом P, являющимся суффиксом T_i. Цикл **while** в строках 6 и 7 выполняет итерации по значениям в $\pi^*[q']$, но хотя $P_q \sqsupset T_{i-1}$ для каждого $q \in \pi^*[q']$, цикл никогда не находит q, такое, что $P[q + 1] = T[i]$. Цикл завершается, когда q достигает нуля, и, конечно, строка 9 не выполняется. Следовательно, $q = 0$ в строке 10, так что $q = \sigma(T_i)$.
- Если $\sigma(T_i) = q' + 1$, то $P[q' + 1] = T[i]$, и проверяемое в строке 6 цикла **while** условие при первом проходе оказывается ложным. Выполняется строка 9 и увеличивается значение q, так что после этого мы имеем $q = q' + 1 = \sigma(T_i)$.
- Если $0 < \sigma(T_i) \leq q'$, то цикл **while** в строках 6 и 7 выполняет по меньшей мере одну итерацию, проверяя в порядке убывания каждое значение $q \in \pi^*[q']$ до тех пор, пока не прекратит работу при некотором $q < q'$. Таким образом, P_q является наидлиннейшим префиксом $P_{q'}$, для которого $P[q+1] = T[i]$, так что, когда цикл **while** завершает работу, $q+1 = \sigma(P_{q'}T[i])$. Поскольку $q' = \sigma(T_{i-1})$, из леммы 32.3 следует, что $\sigma(T_{i-1}T[i]) = \sigma(P_{q'}T[i])$. Таким образом, мы имеем

$$\begin{aligned} q + 1 &= \sigma(P_{q'}T[i]) \\ &= \sigma(T_{i-1}T[i]) \\ &= \sigma(T_i) \end{aligned}$$

  при завершении цикла **while**. После того как строка 9 увеличивает значение q, имеем $q = \sigma(T_i)$.

Строка 12 в процедуре KMP-MATCHER является необходимой, поскольку в противном случае можно обратиться к $P[m + 1]$ в строке 6 после того, как будет найдено вхождение P. (Рассуждение, что $q = \sigma(T_{i-1})$ при следующем выполнении строки 6, остается корректным в силу указания из упр. 32.4.8: $\delta(m, a) = \delta(\pi[m], a)$, или, что эквивалентно, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ для любого $a \in \Sigma$.) Оставшаяся часть доказательства корректности алгоритма Кнута–Морриса–Пратта следует из корректности процедуры FINITE-AUTOMATON-MATCHER, поскольку мы уже показали, что процедура KMP-MATCHER имитирует поведение FINITE-AUTOMATON-MATCHER.

Упражнения

32.4.1

Вычислите префиксную функцию для образца `ababbabbabbababbabb`.

32.4.2

Найдите верхнюю границу размера $\pi^*[q]$ как функцию от величины q. Приведите пример, показывающий, что ваша оценка не может быть улучшена.

32.4.3

Объясните, как найти вхождения образца P в текст T, зная функцию π для PT (т.е. для строки длиной $m + n$, полученной в результате конкатенации строк P и T).

32.4.4

Воспользуйтесь групповым анализом, чтобы показать, что время работы процедуры KMP-MATCHER равно $\Theta(n)$.

32.4.5

Воспользуйтесь функцией потенциала, чтобы показать, что время работы процедуры KMP-MATCHER равно $\Theta(n)$.

32.4.6

Покажите, как улучшить процедуру KMP-MATCHER, заменив функцию π в строке 7 (но не в строке 12) функцией π', рекурсивно определяемой для $q = 1, 2, \ldots, m - 1$ следующим образом:

$$\pi'[q] = \begin{cases} 0 , & \text{если } \pi[q] = 0 , \\ \pi'[\pi[q]] , & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q] + 1] = P[q + 1] , \\ \pi[q] , & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q] + 1] \neq P[q + 1] . \end{cases}$$

Объясните, почему модифицированный алгоритм работает корректно, а также в чем состоит смысл этого улучшения.

32.4.7

Разработайте алгоритм, который в течение линейного времени позволил бы определить, является ли текстовая строка T циклическим сдвигом другой строки T'. Например, строки *arc* и *car* являются циклическими сдвигами одна другой.

32.4.8 ★

Разработайте эффективный алгоритм вычисления функции переходов δ для конечного автомата поиска заданного образца P. Время работы алгоритма должно быть равно $O(m\,|\Sigma|)$. (*Указание:* докажите, что $\delta(q, a) = \delta(\pi[q], a)$, если $q = m$ или $P[q + 1] \neq a$.)

Задачи

32.1. Поиск подстрок на основе коэффициентов повторения

Пусть y^i обозначает строку, полученную конкатенацией i строк y. Например, $(\mathtt{ab})^3 = \mathtt{ababab}$. Говорят, что строка $x \in \Sigma^*$ имеет ***коэффициент повторения*** (repetition factor) r, если $x = y^r$ для некоторой строки $y \in \Sigma^*$ и некоторого $r > 0$. Пусть $\rho(x)$ обозначает наибольшее r, такое, что x имеет коэффициент повторения r.

а. Разработайте эффективный алгоритм, принимающий в качестве входных данных образец $P[1..m]$ и вычисляющий значение $\rho(P_i)$ для $i = 1, 2, \ldots, m$. Чему равно время работы этого алгоритма?

б. Определим для произвольного образца $P[1..m]$ величину $\rho^*(P)$ как $\max_{1 \le i \le m} \rho(P_i)$. Докажите, что если образец P выбирается случайным образом из множества всех бинарных строк длиной m, то математическое ожидание $\rho^*(P)$ равно $O(1)$.

в. Докажите, что приведенный ниже алгоритм поиска подстрок корректно находит все вхождения образца P в текст $T[1..n]$ за время $O(\rho^*(P)n + m)$.

```
REPETITION-MATCHER(P, T)
m = P.length
n = T.length
k = 1 + ρ*(P)
q = 0
s = 0
while s ≤ n − m
    if T[s + q + 1] == P[q + 1]
        q = q + 1
        if q == m
            print "Образец найден со сдвигом" s
    if q == m или T[s + q + 1] ≠ P[q + 1]
        s = s + max(1, ⌈q/k⌉)
        q = 0
```

Этот алгоритм был предложен Галилом (Galil) и Сейферасом (Seiferas). Развивая эти идеи, они получили алгоритм поиска подстрок с линейным временем работы, использующий всего $O(1)$ памяти, кроме необходимой для хранения строк P и T.

Заключительные замечания

Связь поиска подстрок с теорией конечных автоматов обсуждается в книге Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [5]. Алгоритм Кнута–Морриса–Пратта [213] разработан Кнутом (Knuth) и Праттом (Pratt) и независимо Моррисом (Morris); результаты своих исследований они опубликовали совместно. Рейнгольд (Reingold), Урбан (Urban) и Грис (Gries) [292] привели альтернативную трактовку алгоритма Кнута–Морриса–Пратта. Алгоритм Рабина–Карпа был предложен Рабином (Rabin) и Карпом (Karp) [200]. Галил (Galil) и Сейферас (Seiferas) [125] разработали интересный детерминистический алгоритм поиска подстрок с линейным временем работы, в котором используется лишь $O(1)$ памяти сверх необходимой для хранения образца и текста.

Глава 33. Вычислительная геометрия

Вычислительная геометрия — это раздел информатики, изучающий алгоритмы, предназначенные для решения геометрических задач. В современных инженерных и математических расчетах вычислительная геометрия, в числе других областей знаний, применяется в машинной графике, в робототехнике, при разработке СБИС, при автоматизированном проектировании, в металлургии, в статистике... Увы, книга слишком ограничена в объеме для полного перечисления. Роль входных данных в задачах вычислительной геометрии обычно играет описание множества таких геометрических объектов, как точки, отрезки или вершины многоугольника в порядке обхода против часовой стрелки. На выходе часто дается ответ на такие запросы об этих объектах, как наличие пересекающихся линий или параметры новых геометрических объектов, например выпуклой оболочки множества точек (это минимальный выпуклый многоугольник, содержащий данное множество).

В этой главе мы ознакомимся с несколькими алгоритмами вычислительной геометрии в двух измерениях, т.е. на плоскости. Каждый входной объект представлен множеством точек $\{p_1, p_2, p_3, \ldots\}$, где каждая точка $p_i = (x_i, y_i)$ образована парой действительных чисел $x_i, y_i \in \mathbb{R}$. Например, n-угольник P представлен последовательностью вершин $\langle p_0, p_1, p_2, \ldots, p_{n-1}\rangle$ в порядке обхода границы многоугольника P. Вычислительная геометрия может работать в трех измерениях и даже в большем их числе, но визуализация подобных задач и их решений может оказаться очень трудной. Однако даже в двух измерениях можно ознакомиться с хорошими примерами применения методов вычислительной геометрии.

В разделе 33.1 показано, как быстро и точно ответить на такие основные вопросы о расположении отрезков: если два отрезка имеют общую конечную точку, то как следует перемещаться при переходе от одного из них к другому — по часовой стрелке или против часовой стрелки, в каком направлении следует сворачивать при переходе от одного такого отрезка к другому, а также пересекаются ли два отрезка. В разделе 33.2 представлен метод, известный под названием "выметание" (sweeping) или "метод движущейся прямой". Он используется при разработке алгоритма со временем работы $O(n \lg n)$, определяющего, имеются ли пересечения среди n отрезков. В разделе 33.3 приведены два алгоритма "выметания по кругу", предназначенные для вычисления выпуклой оболочки множества n точек (наименьшего содержащего их выпуклого многоугольника): сканирование по Грэхему (Graham's scan), время работы которого равно $O(n \lg n)$, и обход по

Джарвису (Jarvis's march), время выполнения которого равно $O(nh)$, где h – количество вершин в оболочке. Наконец в разделе 33.4 приводится алгоритм на основе парадигмы "разделяй и властвуй" со временем работы $O(n \lg n)$, предназначенный для поиска пары ближайших точек в заданном на плоскости множестве из n точек.

33.1. Свойства отрезков

В некоторых представленных в этой главе алгоритмах требуется ответить на вопросы о свойствах отрезков. ***Выпуклой комбинацией*** (convex combination) двух различных точек $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ называется любая точка $p_3 = (x_3, y_3)$, такая, что для некоторого значения α, принадлежащего интервалу $0 \leq \alpha \leq 1$, выполняются равенства $x_3 = \alpha x_1 + (1 - \alpha) x_2$ и $y_3 = \alpha y_1 + (1 - \alpha) y_2$ (пишут также $p_3 = \alpha p_1 + (1 - \alpha) p_2$). Интуитивно понятно, что в роли p_3 может выступать любая точка, которая принадлежит прямой, соединяющей точки p_1 и p_2, и находится между этими точками. Если заданы две различные точки, p_1 и p_2, то ***отрезком*** (line segment) $\overline{p_1 p_2}$ называется множество выпуклых комбинаций p_1 и p_2. Точки p_1 и p_2 называются ***конечными точками*** (endpoints) отрезка $\overline{p_1 p_2}$. Иногда играет роль порядок следования точек p_1 и p_2, и тогда говорят о ***направленном отрезке*** (directed segment) $\overrightarrow{p_1 p_2}$. Если точка p_1 совпадает с ***началом координат*** (origin), т.е. имеет координаты $(0, 0)$, то направленный отрезок $\overrightarrow{p_1 p_2}$ можно рассматривать как ***вектор*** (vector) p_2.

В этом разделе исследуются перечисленные ниже вопросы.

1. Если заданы два направленных отрезка, $\overrightarrow{p_0 p_1}$ и $\overrightarrow{p_0 p_2}$, то находится ли отрезок $\overrightarrow{p_0 p_1}$ в направлении по часовой стрелке от отрезка $\overrightarrow{p_0 p_2}$ относительно их общей точки p_0?
2. Если заданы два отрезка, $\overline{p_0 p_1}$ и $\overline{p_1 p_2}$, то в какую сторону следует свернуть в точке p_1 при переходе от отрезка $\overline{p_0 p_1}$ к отрезку $\overline{p_1 p_2}$?
3. Пересекаются ли отрезки $\overline{p_1 p_2}$ и $\overline{p_3 p_4}$?

На рассматриваемые точки не накладывается никаких ограничений.

На каждый из этих вопросов можно ответить за время $O(1)$, что не должно вызывать удивления, поскольку объем входных данных, которыми выражается каждый из этих вопросов, равен $O(1)$. Кроме того, в наших методах используются только операции сложения, вычитания, умножения и сравнения. Не понадобится ни деление, ни вычисление тригонометрических функций, а ведь обе эти операции могут оказаться дорогостоящими и приводить к проблемам, связанным с ошибками округления. Например, метод "в лоб" при определении того, пересекаются ли два отрезка, – найти для каждого из них уравнение прямой в виде $y = mx + b$ (где m – коэффициент наклона, а b – координата пересечения с осью y), вычислить точку пересечения этих прямых и проверить, принадлежит ли эта точка обоим отрезкам. В таком методе при вычислении координат точки пересечения используется деление. Если отрезки почти параллельны, этот метод

очень чувствителен к точности операции деления на реальных компьютерах. Изложенный в данном разделе метод, в котором удается избежать операции деления, намного точнее.

Векторное произведение

Вычисление векторного произведения составляет основу методов работы с отрезками. Рассмотрим векторы p_1 и p_2, показанные на рис. 33.1, (а). ***Векторное произведение*** (cross product) $p_1 \times p_2$ можно интерпретировать как знаковое значение площади параллелограмма, образованного точками $(0,0)$, p_1, p_2 и $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Эквивалентное, но более полезное определение векторного произведения — определитель матрицы[1]:

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 . \end{aligned}$$

Если величина $p_1 \times p_2$ положительна, то вектор p_1 находится по часовой стрелке от вектора p_2 относительно начала координат $(0,0)$; если же векторное произведение отрицательно, то вектор p_1 находится в направлении против часовой стрелки от вектора p_2. (См. упр. 33.1.1.) На рис. 33.1, (б) показаны области, расположенные по часовой стрелке и против нее относительно вектора p. Граничный случай возникает, когда векторное произведение равно нулю. В этом случае векторы ***коллинеарны*** (collinear), т.е. направлены в одном и том же или в противоположных направлениях.

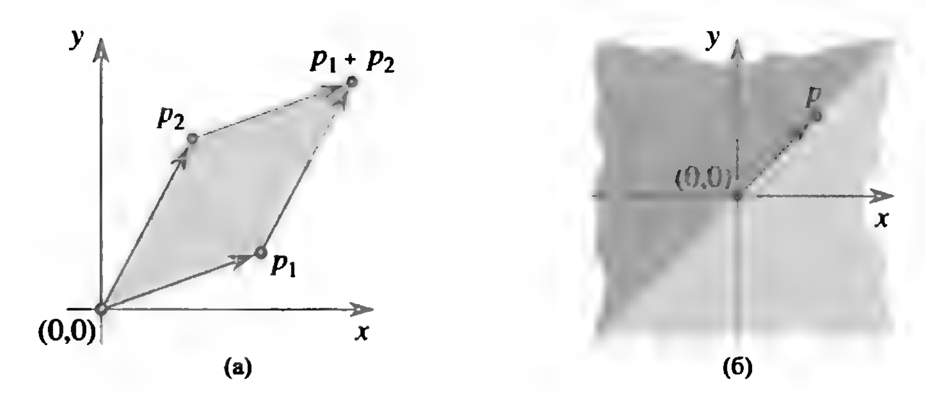


Рис. 33.1. **(а)** Векторное произведение векторов p_1 и p_2 представляет собой площадь параллелограмма со знаком. **(б)** Область со светлой штриховкой содержит векторы, находящиеся от p по часовой стрелке. Область с темной штриховкой содержит векторы, находящиеся от p против часовой стрелки.

---

[1]На самом деле векторное произведение — трехмерная концепция. Это вектор, перпендикулярный векторам p_1 и p_2, направление которого определяется правилом правой руки, а величина равна $|x_1 y_2 - x_2 y_1|$. Однако в этой главе удобнее трактовать векторное произведение просто как величину $x_1 y_2 - x_2 y_1$.

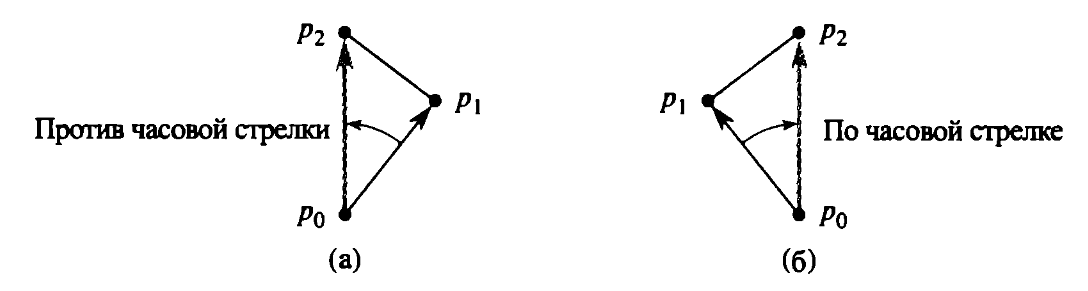


Рис. 33.2. Использование векторного произведения для определения направления поворота последовательных отрезков $\overline{p_0p_1}$ и $\overline{p_1p_2}$ в точке p_1 (выясняем, повернут ли направленный отрезок $\overrightarrow{p_0p_2}$ по часовой стрелке или против нее относительно направленного отрезка $\overrightarrow{p_0p_1}$). **(а)** В случае направления против часовой стрелки выполняется поворот влево. **(б)** В случае направления по часовой стрелке выполняется поворот вправо.

Чтобы определить, находится ли направленный отрезок $\overrightarrow{p_0p_1}$ по часовой стрелке от направленного отрезка $\overrightarrow{p_0p_2}$ или против относительно их общей точки p_0, достаточно использовать ее как начало координат. Сначала обозначим величину $p_1 - p_0$ как вектор $p'_1 = (x'_1, y'_1)$, где $x'_1 = x_1 - x_0$ и $y'_1 = y_1 - y_0$, а затем введем аналогичные обозначения для величины $p_2 - p_0$. После этого вычислим векторное произведение

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \; .$$

Если это векторное произведение положительно, $\overrightarrow{p_0p_1}$ находится по часовой стрелке от $\overrightarrow{p_0p_2}$; если отрицательно — против часовой стрелки.

Поворот последовательных отрезков

Следующий вопрос заключается в том, куда сворачивают два последовательных отрезка, $\overline{p_0p_1}$ и $\overline{p_1p_2}$, в точке p_1 — влево или вправо. Можно привести эквивалентную формулировку этого вопроса — определить знак угла $\angle p_0p_1p_2$. Векторное произведение позволяет ответить на этот вопрос, не вычисляя величину угла. Как видно из рис. 33.2, мы просто проверяем, находится ли направленный отрезок $\overrightarrow{p_0p_2}$ по часовой стрелке или против часовой стрелки относительно направленного отрезка $\overrightarrow{p_0p_1}$. Для этого вычисляется векторное произведение $(p_2 - p_0) \times (p_1 - p_0)$. Если эта величина отрицательна, то направленный отрезок $\overrightarrow{p_0p_2}$ находится против часовой стрелки по отношению к направленному отрезку $\overrightarrow{p_0p_1}$, так что в точке p_1 мы делаем поворот влево. Положительное значение векторного произведения указывает на ориентацию по часовой стрелке и поворот вправо. Нулевое векторное произведение означает, что точки p_0, p_1 и p_2 коллинеарны.

Определение, пересекаются ли два отрезка

Чтобы определить, пересекаются ли два отрезка, следует проверить, пресекает ли каждый из них прямую, содержащую другой отрезок. Отрезок $\overline{p_1p_2}$ ***пересекает*** (straddles) прямую, если конечные точки отрезка p_1 и p_2 лежат в разных полуплоскостях, на которые прямая разбивает плоскость. В граничном случае точка p_1 или точка p_2 (или обе эти точки) лежит непосредственно на прямой. Два отрезка

пересекаются тогда и только тогда, когда выполняется одно из сформулированных ниже условий (или оба эти условия одновременно).

1. Каждый отрезок пересекает прямую, на которой лежит другой отрезок.
2. Конечная точка одного из отрезков лежит на другом отрезке (граничный случай).

Эта идея реализована в приведенных ниже процедурах. Процедура SEGMENTS-INTERSECT возвращает значение TRUE, если отрезки $\overline{p_1p_2}$ и $\overline{p_3p_4}$ пересекаются, и значение FALSE — в противном случае. В этой процедуре вызываются вспомогательные процедуры DIRECTION и ON-SEGMENT. В первой из них с помощью описанного выше векторного произведения определяется относительное расположение отрезков, а во второй — лежит ли на отрезке точка, если известно, что она коллинеарна этому отрезку.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
1  d1 = DIRECTION(p3, p4, p1)
2  d2 = DIRECTION(p3, p4, p2)
3  d3 = DIRECTION(p1, p2, p3)
4  d4 = DIRECTION(p1, p2, p4)
5  if ((d1 > 0 и d2 < 0) или (d1 < 0 и d2 > 0)) и
        ((d3 > 0 и d4 < 0) или (d3 < 0 и d4 > 0))
6       return TRUE
7  elseif d1 == 0 и ON-SEGMENT(p3, p4, p1)
8       return TRUE
9  elseif d2 == 0 и ON-SEGMENT(p3, p4, p2)
10      return TRUE
11 elseif d3 == 0 и ON-SEGMENT(p1, p2, p3)
12      return TRUE
13 elseif d4 == 0 и ON-SEGMENT(p1, p2, p4)
14      return TRUE
15 else return FALSE
```

```
DIRECTION(pi, pj, pk)
1  return (pk − pi) × (pj − pi)
```

```
ON-SEGMENT(pi, pj, pk)
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) и min(yi, yj) ≤ yk ≤ max(yi, yj)
2       return TRUE
3  else return FALSE
```

Процедура SEGMENTS-INTERSECT работает следующим образом. В строках 1–4 вычисляется относительное расположение d_i каждой конечной точки p_i относительно другого отрезка. Если все величины, характеризующие относительное расположение, отличны от нуля, то легко определить, пересекаются ли отрезки $\overline{p_1p_2}$ и $\overline{p_3p_4}$. Это делается следующим образом. Отрезок $\overline{p_1p_2}$ пересекает прямую, содержащую отрезок $\overline{p_3p_4}$, если направленные отрезки $\overrightarrow{p_3p_1}$ и $\overrightarrow{p_3p_2}$ имеют

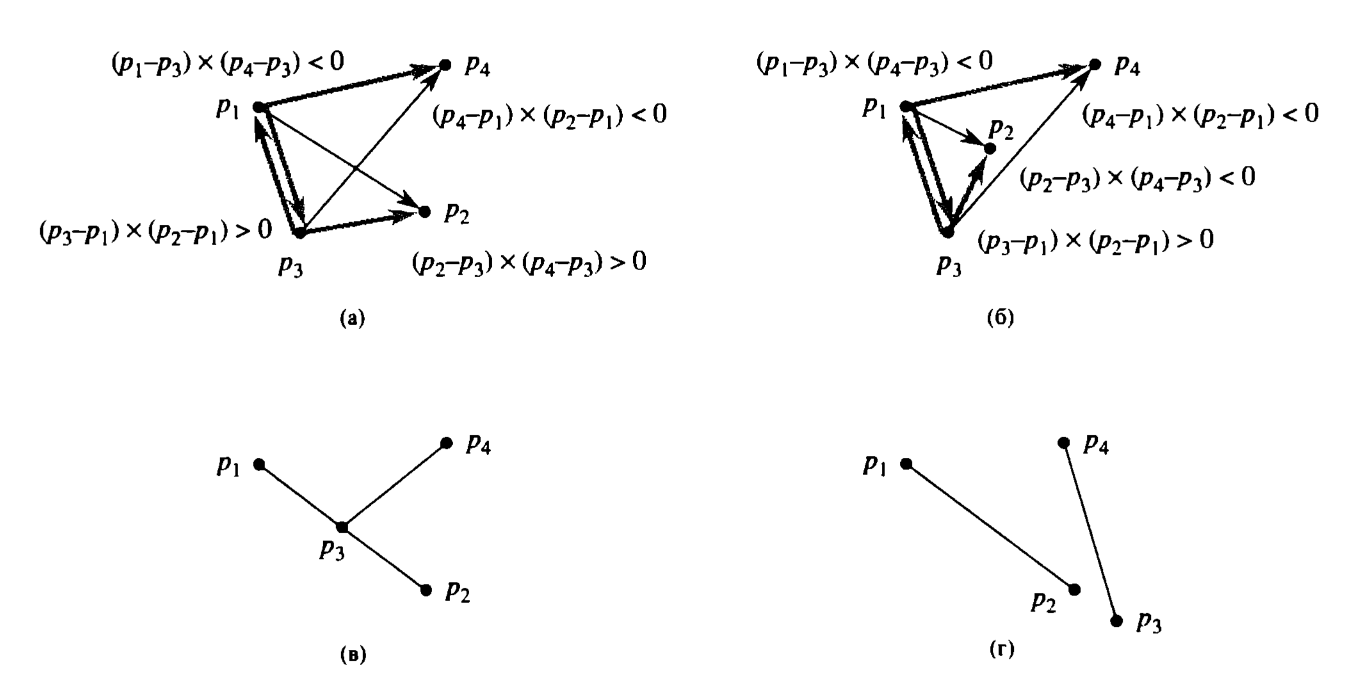


Рис. 33.3. Частные случаи в процедуре SEGMENTS-INTERSECT. **(а)** Отрезки $\overline{p_1p_2}$ и $\overline{p_3p_4}$ пересекают прямые друг друга. Поскольку $\overline{p_3p_4}$ пересекает прямую, содержащую $\overline{p_1p_2}$, знаки векторных произведений $(p_3 - p_1) \times (p_2 - p_1)$ и $(p_4 - p_1) \times (p_2 - p_1)$ различны. Поскольку $\overline{p_1p_2}$ пересекает прямую, содержащую $\overline{p_3p_4}$, знаки векторных произведений $(p_1 - p_3) \times (p_4 - p_3)$ и $(p_2 - p_3) \times (p_4 - p_3)$ различны. **(б)** Отрезок $\overline{p_3p_4}$ пересекает прямую, содержащую $\overline{p_1p_2}$, но отрезок $\overline{p_1p_2}$ не пересекает прямую, содержащую $\overline{p_3p_4}$. Знаки векторных произведений $(p_1 - p_3) \times (p_4 - p_3)$ и $(p_2 - p_3) \times (p_4 - p_3)$ одинаковы. **(в)** Точка p_3 коллинеарна отрезку $\overline{p_1p_2}$ и находится между p_1 и p_2. **(г)** Точка p_3 коллинеарна отрезку $\overline{p_1p_2}$, но не находится между p_1 и p_2. Отрезки не пересекаются.

противоположные направления относительно $\overrightarrow{p_3p_4}$. В этом случае знаки величин d_1 и d_2 разные. Аналогично отрезок $\overline{p_3p_4}$ пересекает прямую, содержащую отрезок $\overline{p_1p_2}$, если знаки величин d_3 и d_4 разные. Если проверка в строке 5 успешна, то отрезки пересекаются, и процедура SEGMENTS-INTERSECT возвращает значение TRUE. Этот случай показан на рис. 33.3, (а). В противном случае отрезки не пересекают прямые друг друга, хотя и возможны граничные случаи. Если ни одна из величин, характеризующих взаимную ориентацию отрезков, не равна нулю, то это не граничный случай. При этом не выполняется ни одно из условий равенства нулю в строках 7–13, и процедура SEGMENTS-INTERSECT возвращает в строке 15 значение FALSE. Этот случай проиллюстрирован на рис. 33.3, (б).

Граничный случай осуществляется, когда любая относительная ориентация d_k равна нулю. Это говорит о том, что точка p_k коллинеарна с другим отрезком. Данная точка принадлежит другому отрезку тогда и только тогда, когда она находится между его конечными точками. Процедура ON-SEGMENT позволяет определить, расположена ли точка p_k между конечными точками другого отрезка $\overline{p_ip_j}$. Эта процедура вызывается в строках 7–13, и в ней предполагается, что точка p_k коллинеарна отрезку $\overline{p_ip_j}$. В частях (в) и (г) рис. 33.3 проиллюстрирован случай, когда один из отрезков коллинеарен конечной точке другого отрезка. В части (в) точка p_3 лежит на отрезке $\overline{p_1p_2}$, так что процедура SEGMENTS-INTERSECT возвращает в строке 12 значение TRUE. В части (г) точка p_3 коллинеарна отрезку $\overline{p_1p_2}$,

но не лежит между его конечными точками. Процедура SEGMENTS-INTERSECT возвращает в строке 15 значение FALSE (отрезки не пересекаются).

Другие применения векторного произведения

В последующих разделах этой главы описаны другие приложения векторного произведения. В разделе 33.3 ставится задача сортировки множества точек по характеризующему их расположение полярным углам относительно заданного начала координат. В упр. 33.1.3 предлагается показать, что с помощью векторного произведения можно осуществить сравнение в соответствующей процедуре сортировки. В разделе 33.2 с помощью красно-черных деревьев поддерживается упорядочение отрезков по вертикали. Вместо того чтобы явным образом поддерживать ключевые значения, в коде, реализующем красно-черное дерево, их сравнение будет заменено векторным произведением. Это позволит определить, какой из двух отрезков, пересекающих заданную прямую, находится выше другого.

Упражнения

33.1.1

Докажите, что если произведение $p_1 \times p_2$ положительно, то переход от вектора p_2 к вектору p_1 относительно начала координат $(0, 0)$ осуществляется по часовой стрелке, а если это векторное произведение отрицательно, то переход осуществляется против часовой стрелки.

33.1.2

Профессор предположил, что в строке 1 процедуры ON-SEGMENT достаточно протестировать только измерение x. Покажите, что профессор ошибается.

33.1.3

Полярным углом (polar angle) точки p_1 относительно начала координат p_0 называется угол между вектором $p_1 - p_0$ в обычной полярной системе координат. Например, полярный угол точки $(3, 5)$ относительно точки $(2, 4)$ — это угол вектора $(1, 1)$, составляющий 45° или $\pi/4$ радиан. Полярный угол точки $(3, 3)$ относительно точки $(2, 4)$ — это угол вектора $(1, -1)$, составляющий угол 315° или $7\pi/4$ радиан. Напишите псевдокод, сортирующий последовательность n точек $\langle p_1, p_2, \ldots, p_n \rangle$ по их полярному углу относительно заданной точки p_0. Процедура должна выполняться за время $O(n \lg n)$, и сравнение углов в ней следует выполнять с помощью векторного произведения.

33.1.4

Покажите, как за время $O(n^2 \lg n)$ определить, содержатся ли в множестве из n точек три коллинеарные точки.

33.1.5

Многоугольник (polygon) представляет собой кусочно-линейную замкнутую кривую на плоскости. Другими словами, это образованная последовательностью отрезков кривая, начало которой совпадает с ее концом. Эти отрезки называют-

ся ***сторонами*** (sides) многоугольника. Точка, соединяющая две последовательные стороны, называется ***вершиной*** (vertex) многоугольника. Если многоугольник ***простой*** (simple), в нем нет самопересечений. (В общем случае предполагается, что мы имеем дело с простыми многоугольниками.) Множество точек плоскости, ограниченное простым многоугольником, образует ***внутреннюю область*** (interior) многоугольника, множество точек, принадлежащих самому многоугольнику, образует его ***границу*** (boundary), а множество точек, окружающих многоугольник, образует его ***внешнюю область*** (exterior). Простой многоугольник называется ***выпуклым*** (convex), если отрезок, соединяющий любые две его граничные или внутренние точки, лежит на границе или во внутренней части этого многоугольника.

Профессор предложил метод, позволяющий определить, являются ли n точек $\langle p_0, p_1, \ldots, p_{n-1}\rangle$ последовательными вершинами выпуклого многоугольника. Предложенный алгоритм выводит положительный ответ, если в множестве $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, где увеличение индексов выполняется по модулю n, не содержатся одновременно и левые, и правые повороты. В противном случае выводится отрицательный ответ. Покажите, что несмотря на то, что время работы этой процедуры выражается линейной функцией, она не всегда дает правильный ответ. Модифицируйте предложенный профессором метод, чтобы он всегда давал правильный ответ за линейное время.

33.1.6

Пусть задана точка $p_0 = (x_0, y_0)$. ***Правым горизонтальным лучом*** (right horizontal ray) из точки p_0 называется множество точек $\{p_i = (x_i, y_i) : x_i \geq x_0$ и $y_i = y_0\}$, т.е. множество точек, расположенных строго справа от точки p_0, вместе с самой этой точкой. Покажите, как за время $O(1)$ определить, пересекает ли данный правый горизонтальный луч из точки p_0 отрезок $\overline{p_1 p_2}$. Сведите эту задачу к другой, в которой определяется, пересекаются ли два отрезка.

33.1.7

Один из способов определения, находится ли заданная точка p_0 во внутренней области простого, но не обязательного выпуклого многоугольника P, заключается в следующем. Из точки p_0 проводится произвольный луч и определяется, сколько раз он пересекает границу многоугольника P. Если количество пересечений нечетно и сама точка p_0 не лежит на границе многоугольника P, то она лежит во внутренней его части. Покажите, как в течение времени $\Theta(n)$ определить, находится ли точка p_0 во внутренней части n-угольника P. (*Указание:* воспользуйтесь результатами упр. 33.1.6. Убедитесь в правильности этого алгоритма в том случае, когда луч пересекает границу многоугольника в его вершине и если луч перекрывается его стороной.)

33.1.8

Покажите, как за время $\Theta(n)$ найти площадь простого, но не обязательно выпуклого n-угольника. (Определения, которые относятся к многоугольникам, можно найти в упр. 33.1.5.)

33.2. Определение наличия пересекающихся отрезков

В этом разделе представлен алгоритм для определения, пересекаются ли какие-нибудь два из отрезков некоторого множества. В этом алгоритме используется метод, известный под названием "выметание", который часто встречается в алгоритмах вычислительной геометрии. Кроме того, как показано в упражнениях, приведенных в конце этого раздела, с помощью данного алгоритма или его вариации можно решать и другие задачи вычислительной геометрии.

Время работы этого алгоритма равно $O(n \lg n)$, где n — количество заданных отрезков. В нем лишь определяется, существуют пересечения или нет, но не выводятся данные обо всех этих пересечениях. (Согласно результатам упр. 33.2.1 для поиска *всех* пересечений в множестве, состоящем из n отрезков, в наихудшем случае понадобится время $\Omega(n^2)$.)

В методе ***выметания*** (sweeping) по заданному множеству геометрических объектов проводится воображаемая вертикальная ***выметающая прямая*** (sweep line), которая обычно движется слева направо. Измерение, вдоль которого двигается выметающая прямая (в данном случае это измерение x), трактуется как время. Выметание предоставляет способ упорядочения геометрических объектов, обычно путем размещения их параметров в динамической структуре данных, что позволяет воспользоваться взаимоотношениями между этими объектами. В приведенном в этом разделе алгоритме, устанавливающем наличие пересечений отрезков, рассматриваются все конечные точки отрезков в порядке их расположения слева направо, и для каждой новой конечной точки проверяется наличие пересечений.

Чтобы описать алгоритм и доказать его способность корректно определить, пересекаются ли какие-либо из n отрезков, сделаем два упрощающих предположения. Предположим, во-первых, что ни один из входных отрезков не является вертикальным, и во-вторых — что никакие три входных отрезка не пересекаются в одной точке. В упр. 33.2.8 и 33.2.9 предлагается показать, что алгоритм достаточно надежен для того, чтобы после небольших изменений работать даже в тех случаях, когда сделанные предположения не выполняются. Часто оказывается, что отказ от подобных упрощений и учет граничных условий становится самым сложным этапом программирования алгоритмов вычислительной геометрии и доказательства их корректности.

Упорядочение отрезков

Поскольку предполагается, что вертикальные отрезки отсутствуют, каждый входной отрезок пересекает данную вертикальную выметающую прямую в одной точке. Таким образом, отрезки, пересекающие вертикальную выметающую прямую, можно упорядочить по координате y точки пересечения.

Чтобы быть более точными, рассмотрим два отрезка, s_1 и s_2. Говорят, что они ***сравнимы*** (comparable) в координате x, если вертикальная выметающая прямая с координатой x пересекает оба этих отрезка. Говорят также, что отрезок s_1 расположен ***над*** (above) отрезком s_2 в x (записывается $s_1 \succcurlyeq_x s_2$), если отрезки s_1

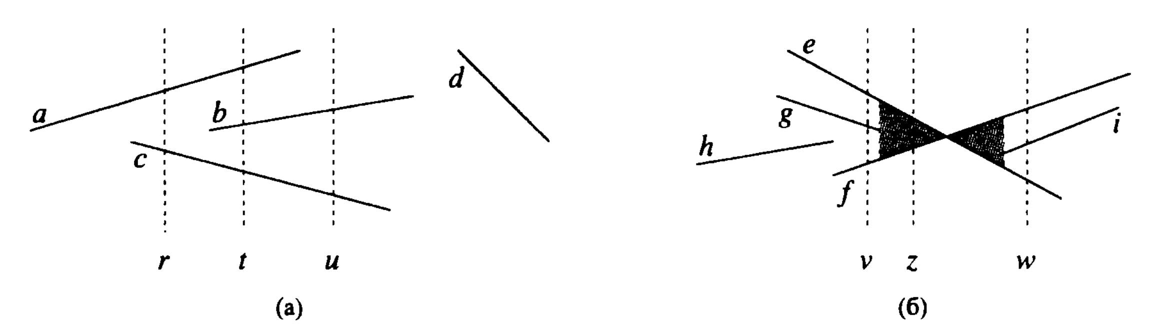


Рис. 33.4. Упорядочение отрезков с помощью разных вертикальных выметающих прямых. **(а)** Здесь $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$ и $b \succcurlyeq_u c$. Отрезок d не сравним ни с одним из показанных отрезков. **(б)** Когда отрезки e и f пересекаются, их порядок становится обратным: $e \succcurlyeq_v f$, но $f \succcurlyeq_w e$. Для любой выметающей прямой (такой, как z), которая проходит по заштрихованной области, отрезки e и f являются последовательными при упорядочении отношением $\succcurlyeq_z$.

и s_2 сравнимы в координате x и точка пересечения отрезка s_1 с выметающей прямой в координате x находится выше, чем точка пересечения отрезка s_2 с этой выметающей прямой, или если s_1 и s_2 пересекаются на выметающей прямой. Например, в ситуации, проиллюстрированной на рис. 33.4, (а), выполняются соотношения $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$ и $b \succcurlyeq_u c$. Отрезок d не сравним ни с каким другим отрезком.

Для произвольной заданной координаты x отношение "$\succcurlyeq_x$" полностью упорядочивает (см. раздел Б.2) отрезки, пересекающие выметающую прямую в координате x. Иначе говоря, это отношение транзитивно, и если каждый из отрезков s_1 и s_2 пересекает выметающую прямую в x, то справедливо либо отношение $s_1 \succcurlyeq_x s_2$, либо отношение $s_2 \succcurlyeq_x s_1$, либо они оба (если s_1 и s_2 пересекаются на выметающей прямой). (Отношение $\succcurlyeq_x$ является также рефлексивным, но ни симметричным, ни антисимметричным.) Однако общий порядок может отличаться для разных значений x по мере входа отрезков в число сравнимых и выхода из него. Отрезок попадает в число сравнимых, когда выметающая прямая проходит его левую конечную точку, и выходит из него, когда выметающая прямая проходит его правую конечную точку.

Что происходит, когда выметающая прямая проходит через точку пересечения двух отрезков? Как видно из рис. 33.4, (б), в общем упорядочении меняется порядок этих отрезков. В показанной на рисунке ситуации выметающие прямые v и w находятся соответственно слева и справа от точки пересечения отрезков e и f, и справедливы соотношения $e \succcurlyeq_v f$ и $f \succcurlyeq_w e$. В силу предположения о том, что никакие три отрезка не пересекаются в одной и той же точке, должна существовать некоторая вертикальная выметающая прямая x, для которой пересекающиеся отрезки e и f являются *последовательными* (т.е. между ними нет других отрезков) в полном упорядочении отношением $\succcurlyeq_x$. Для любой выметающей прямой, проходящей через затененную область на рис. 33.4, (б), например для прямой z, отрезки e и f являются последовательными в общем упорядочении.

Перемещение выметающей прямой

В выметающих алгоритмах обычно обрабатываются два набора данных.

1. ***Состояние относительно выметающей прямой*** (sweep-line status) описывает соотношения между объектами, пересекаемыми выметающей прямой.
2. ***Таблица***, или ***расписание, точек-событий*** (event-point schedule) — это последовательность координат x, упорядоченных слева направо, в которой определяются точки останова выметающей прямой. Каждая такая точка останова называется ***точкой-событием*** (event point). По мере перемещения выметающей прямой слева направо, когда она достигает x-координаты точки-события, выметание приостанавливается, выполняется обработка точки-события, после чего выметание продолжается. Изменение состояния относительно выметающей прямой происходит только в точках-событиях.

В некоторых алгоритмах (например, в том, который предлагается разработать в упр. 33.2.7) таблица точек-событий строится динамически, в ходе работы алгоритма. Однако в алгоритме, о котором сейчас пойдет речь, точки-события определяются статически, исходя исключительно из простых свойств входных данных. В частности, в роли точки-события выступает каждая конечная точка отрезка. Конечные точки отрезков сортируются в порядке возрастания координат x (т.е. слева направо). (Если две или более точек ***ковертикальны*** (covertical), т.е. их координаты x совпадают, этот узел можно "разрубить", поместив все ковертикальные левые конечные точки перед ковертикальными правыми конечными точками. Если в множестве ковертикальных левых точек есть несколько левых, первыми среди них будут те, которые имеют меньшее значение координаты y, и то же самое выполняется в множестве ковертикальных правых конечных точек.) Отрезок помещается в набор состояний относительно выметающей прямой, когда пересекается его левая конечная точка; этот отрезок удаляется из набора состояний относительно выметающей прямой, когда пересекается его правая конечная точка. Если два отрезка впервые становятся последовательными в полном упорядочении, выполняется проверка, пересекаются ли они.

Состояние относительно выметающей прямой является полностью упорядоченным множеством T, в котором необходимо реализовать такие операции.

- INSERT(T, s): вставка отрезка s в T.
- DELETE(T, s): удаление отрезка s из T.
- ABOVE(T, s): возврат отрезка, находящегося непосредственно над отрезком s в T.
- BELOW(T, s): возврат отрезка, находящегося непосредственно под отрезком s в T.

Возможна ситуация, когда отрезки s_1 и s_2 находятся взаимно один над другим в упорядочении T; такая ситуация может возникнуть, если s_1 и s_2 пересекаются на выметающей прямой, для которой задается упорядочение T. В этом случае эти два отрезка могут находиться в T в любом порядке.

Если всего имеется n входных отрезков, каждую из перечисленных выше операций с помощью красно-черных деревьев можно выполнить за время $O(\lg n)$. Напомним, что операции над красно-черными деревьями, описанные в главе 13, включают в себя сравнение ключей. Однако в нашем случае сравнение ключей

можно заменить векторным произведением, позволяющим определить относительный порядок двух отрезков (см. упр. 33.2.2).

Псевдокод, выявляющий пересечение отрезков

В приведенном далее алгоритме в качестве входных данных выступает множество S, состоящее из n отрезков. На выходе возвращается булево значение TRUE, если некоторая пара отрезков из множества S пересекается, и значение FALSE — в противном случае. Полное упорядочение T реализуется с помощью красно-черного дерева.

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  Сортировка конечных точек отрезков S слева направо
        с разрешением совпадений путем помещения левых
        конечных точек перед правыми и путем помещения
        точек с меньшими координатами y перед теми,
        координата y которых больше
 3  for каждой точки p в отсортированном списке конечных точек
 4      if p является левой конечной точкой отрезка s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) существует и пересекает s)
                или (BELOW(T, s) существует и пересекает s)
 7              return TRUE
 8      if p является правой конечной точкой отрезка s
 9          if (и ABOVE(T, s), и BELOW(T, s) существуют)
                и (ABOVE(T, s) пересекает BELOW(T, s))
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Работа этого алгоритма проиллюстрирована на рис. 33.5. В строке 1 инициализируется пустое множество полностью упорядоченных отрезков. В строке 2 путем сортировки $2n$ конечных точек отрезков слева направо создается таблица точек-событий с разрешением совпадений, как сказано выше. Одним из способов выполнения строки 2 является лексикографическая сортировка конечных точек (x, e, y), где x и y — обычные координаты, а переменная e принимает значение 0 для левых конечных точек и значение 1 — для правых конечных точек.

Каждая итерация цикла **for** в строках 3–11 обрабатывает одиу точку-событие p. Если p является левой точкой отрезка s, то в строке 5 этот отрезок s добавляется в полностью упорядоченное множество, а в строках 6 и 7 возвращается значение TRUE, если отрезок s пересекает любой из отрезков, соседних с данным в полностью упорядоченном множестве, определяемом выметающей прямой, проходящей через p. (Если точка p находится на другом отрезке s', имеет место граничный случай. При этом требуется лишь, чтобы отрезки s и s' были последовательными в множестве T.) Если p представляет собой правую конечную точку отрезка s, то этот отрезок подлежит удалению из полностью упорядочен-

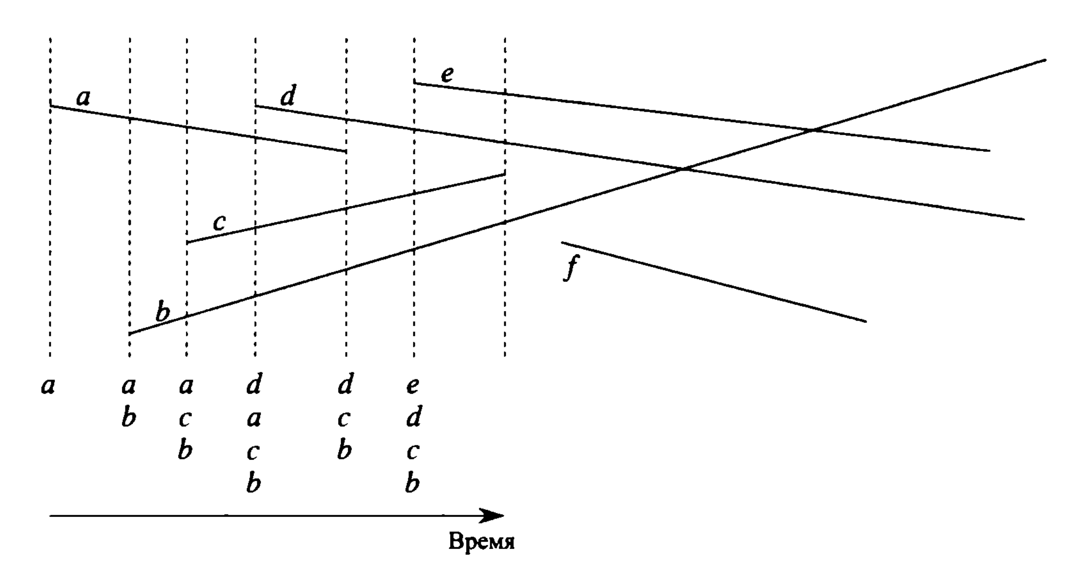


Рис. 33.5. Выполнение процедуры ANY-SEGMENTS-INTERSECT. Каждая пунктирная линия является выметающей прямой, проходящей через одну из точек-событий. За исключением крайней справа выметающей линии упорядочение имен отрезков внизу под каждой выметающей линией соответствует полному упорядочению T в конце цикла **for**, обрабатывающего соответствующую точку-событие. Крайняя справа выметающая прямая соответствует обработке правой конечной точки отрезка c; поскольку отрезки d и b окружают c и пересекают один другой, процедура возвращает TRUE.

ного множества. Но сначала, если пересекаются окружающие s отрезки из полностью упорядоченного множества, определенного выметающей прямой, которая проходит через точку p, в строках 9 и 10 возвращается значение TRUE. Если эти отрезки не пересекаются, в строке 11 отрезок s удаляется из полностью упорядоченного множества. Приведенное ниже доказательство корректности пояснит, почему достаточно проверки отрезков, окружающих s. Наконец, если во всех $2n$ точках-событиях не обнаружено никаких пересечений, в строке 12 возвращается значение FALSE.

Корректность алгоритма

Чтобы показать, что процедура ANY-SEGMENTS-INTERSECT работает корректно, докажем, что при вызове ANY-SEGMENTS-INTERSECT(S) значение TRUE возвращается тогда и только тогда, когда какие-либо отрезки из S пересекаются.

Легко видеть, что процедура ANY-SEGMENTS-INTERSECT возвращает значение TRUE (в строках 7 и 10) только в том случае, когда в ней обнаружено пересечение между двумя входными отрезками. Таким образом, если возвращается значение TRUE, то пересечение существует.

Следует также показать обратное: если пересечение существует, то процедура ANY-SEGMENTS-INTERSECT возвращает значение TRUE. Предположим, что существует по крайней мере одно пересечение. Пусть p — крайняя слева точка пересечения (если таких точек несколько, то выбирается точка с наименьшей координатой y), а a и b — отрезки, пересекающиеся в этой точке. Поскольку слева от точки p нет никаких пересечений, порядок, заданный в множестве T, остается правильным во всех точках, которые находятся слева от точки p. Так как никакие три отрезка не пересекаются в одной и той же точке, существует выметающая

прямая z, относительно которой отрезки a и b расположены последовательно в полностью упорядоченном множестве[2]. Более того, прямая z проходит через точку p или слева от нее. На выметающей прямой z находится конечная точка q какого-нибудь отрезка, которая выступает в роли события, при котором отрезки a и b становятся последовательными в полностью упорядоченном множестве. Если на этой выметающей прямой лежит точка p, то $q = p$. Если же точка p не принадлежит выметающей прямой z, то точка q находится слева от точки p. В любом случае порядок, установленный в множестве T, является правильным непосредственно перед точкой q. (Вот где используется лексикографический порядок, в котором алгоритм обрабатывает конечные точки. Поскольку p — самая низкая из крайних слева левых точек пересечения, то даже если она лежит на выметающей прямой z и на этой прямой есть еще одна точка пересечения p', точка-событие $q = p$ обрабатывается перед тем, как эта другая точка пересечения сможет повлиять на порядок отрезков в упорядоченном множестве T. Более того, даже если p — левая конечная точка одного отрезка (скажем, отрезка a) и правая конечная точка другого отрезка (скажем, отрезка b), то в силу того, что левые точки-события расположены перед правыми, отрезок b уже будет находиться в множестве T, когда в это множество попадет отрезок a.) Каждая точка-событие q либо обрабатывается процедурой ANY-SEGMENTS-INTERSECT, либо нет.

Если точка q обрабатывается процедурой ANY-SEGMENTS-INTERSECT, возможны всего два варианта выполняемых действий.

1. Отрезок a либо отрезок b помещается в множество T, и в этом множестве один из них расположен над или под другим. То, какой из этих случаев имеет место, определяется в строках 4–7.
2. Отрезки a и b уже содержатся в множестве T, а расположенный между ними отрезок удаляется из этого множества, в результате чего отрезки a и b становятся последовательными. Этот случай выявляется в строках 8–11.

В любом случае мы находим пересечение p, и процедура ANY-SEGMENTS-INTERSECT возвращает TRUE.

Если точка-событие q не обрабатывается процедурой ANY-SEGMENTS-INTERSECT, это означает, что произошел выход из процедуры до обработки всех точек-событий. Это может произойти только в том случае, если в этой процедуре уже была обнаружена точка пересечения и процедурой возвращено значение TRUE.

Таким образом, если отрезки пересекаются, то процедура ANY-SEGMENTS-INTERSECT возвращает значение TRUE. Как мы уже убедились, если процедура ANY-SEGMENTS-INTERSECT возвращает значение TRUE, отрезки пересекаются. Поэтому рассматриваемая процедура всегда выдает правильный ответ.

---

[2]Если позволить трем отрезкам пересекаться в одной точке, появится возможность существования промежуточного отрезка c, пересекающего отрезки a и b в точке p. Иначе говоря, для всех выметающих прямых w, расположенных слева от точки p, для которых $a \succcurlyeq_w b$, могут выполняться соотношения $a \succcurlyeq_w c$ и $c \succcurlyeq_w b$. В упр. 33.2.8 предлагается показать, что процедура ANY-SEGMENTS-INTERSECT работает корректно, даже если три отрезка пересекаются в одной и той же точке.

Время работы

Если множество S содержит n отрезков, то процедура ANY-SEGMENTS-INTERSECT выполняется за время $O(n \lg n)$. Выполнение строки 1 занимает время $O(1)$. Строка 2 с помощью сортировки слиянием или пирамидальной сортировки выполняется за время $O(n \lg n)$. Поскольку всего имеется $2n$ точек-событий, а цикл **for** в строках 3–11 выполняет не более одной итерации на точку, он выполняется не более $2n$ раз. На каждую итерацию требуется время $O(\lg n)$, поскольку выполнение каждой операции в красно-черном дереве занимает время $O(\lg n)$, и с помощью метода, описанного в разделе 33.1, каждая проверка пересечения выполняется за время $O(1)$. Таким образом, полное время работы равно $O(n \lg n)$.

Упражнения

33.2.1

Покажите, что в множестве, состоящем из n отрезков, может быть $\Theta(n^2)$ пересечений.

33.2.2

Пусть отрезки a и b сравнимы в координате x. Покажите, как за время $O(1)$ определить, какое из соотношений выполняется — $a \succeq_x b$ или $b \succeq_x a$. Предполагается, что вертикальные отрезки отсутствуют. (*Указание:* если отрезки a и b не пересекаются, достаточно просто воспользоваться векторным произведением. Если же эти отрезки пересекаются, что выявляется путем вычисления векторных произведений, то и в этом случае можно ограничиться только операциями сложения, вычитания и умножения и избежать деления. Если отрезки a и b пересекаются, достаточно просто вывести сообщение о том, что обнаружено пересечение.)

33.2.3

Профессор предложил модифицировать процедуру ANY-SEGMENTS-INTERSECT таким образом, чтобы она не прекращала работу после того, как будет обнаружено пересечение, а выводила пересекающиеся отрезки и переходила к выполнению следующей итерации цикла **for**. Профессор назвал полученную в результате процедуру PRINT-INTERSECTING-SEGMENTS и заявил, что она выводит все пересечения, следующие слева направо в том порядке, в котором они располагаются в множестве отрезков. Аспирант утверждает, что профессор ошибается. Кто из них прав? Всегда ли процедура PRINT-INTERSECTING-SEGMENTS будет находить первым крайнее слева пересечение? Будет ли она находить все пересечения?

33.2.4

Разработайте алгоритм, позволяющий за время $O(n \lg n)$ определить, является ли n-угольник простым.

33.2.5

Разработайте алгоритм, позволяющий за время $O(n \lg n)$ определить, пересекаются ли два простых многоугольника, суммарное количество вершин в которых равно n.

33.2.6

Круг состоит из окружности и ее внутренней области, и его можно представить с помощью центра и радиуса. Два круга пересекаются, если у них есть хотя бы одна общая точка. Приведите алгоритм, позволяющий в течение времени $O(n \lg n)$ определить, пересекаются ли какие-либо два круга из множества, состоящего из n кругов.

33.2.7

Пусть задано множество, состоящее из n отрезков, между которыми имеется k пересечений. Покажите, как вывести данные по всем пересечениям за время $O((n + k) \lg n)$.

33.2.8

Докажите, что процедура ANY-SEGMENTS-INTERSECT работает корректно даже в том случае, если в одной и той же точке пересекается три или более отрезков.

33.2.9

Покажите, что процедура ANY-SEGMENTS-INTERSECT работает корректно даже в том случае, если в числе ее входных отрезков есть вертикальные (при этом нижние конечные точки вертикальных отрезков обрабатываются как левые конечные точки, а верхние — как правые конечные точки). Как изменится ответ в упр. 33.2.2, если допускается наличие вертикальных отрезков?

33.3. Поиск выпуклой оболочки

Выпуклой оболочкой (convex hull) множества точек Q (обозначается $\text{CH}(Q)$) называется наименьший выпуклый многоугольник P, такой, что каждая точка из Q находится либо на границе многоугольника P, либо в его внутренней области. (Точное определение выпуклого многоугольника можно найти в упр. 33.1.5.) Неявно предполагается, что все точки множества Q различны и что в нем имеется как минимум три не коллинеарные точки. Интуитивно можно представлять каждую точку множества Q в виде торчащего из доски гвоздя; тогда выпуклая оболочка будет иметь форму, полученную в результате наматывания на гвозди тугой резиновой нити. Пример множества точек и их выпуклой оболочки приведен на рис. 33.6.

В этом разделе будут представлены два алгоритма, позволяющие построить выпуклую оболочку множества, состоящего из n точек. Оба алгоритма выводят вершины выпуклой оболочки в порядке обхода против часовой стрелки. Время работы первого из них, известного как сканирование по Грэхему, равно $O(n \lg n)$. Второй алгоритм, который называется обходом по Джарвису, выполняется за время $O(nh)$, где h — количество вершин выпуклой оболочки. Как видно из рис. 33.6, каждая вершина $\text{CH}(Q)$ — это точка множества Q. Это свойство используется в обоих алгоритмах. В них принимается решение о том, какие точки из множества Q выступают в роли вершин выпуклой оболочки, а какие следует отбросить.

Вычислить выпуклую оболочку за время $O(n \lg n)$ можно несколькими методами. И при сканировании по Грэхему, и при обходе по Джарвису использует-

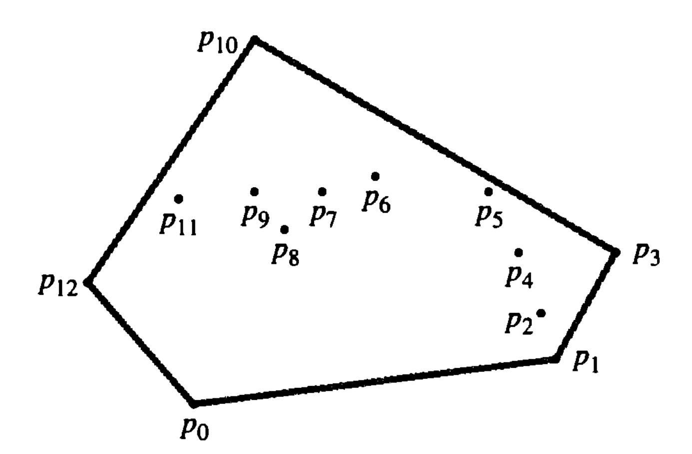


Рис. 33.6. Множество точек $Q = \{p_0, p_1, \ldots, p_{12}\}$ и его выпуклая оболочка $\text{CH}(Q)$.

ся метод, который называется "выметание по кругу". Вершины обрабатываются в порядке возрастания их полярных углов, которые они образуют с некоторой базисной вершиной. В число других методов входят те, которые перечислены ниже.

- В ***инкрементном методе*** (incremental method) точки сначала сортируются слева направо, в результате чего получается последовательность $\langle p_1, p_2, \ldots, p_n \rangle$. На i-м этапе выпуклая оболочка $i-1$ крайних слева точек $\text{CH}(\{p_1, p_2, \ldots, p_{i-1}\})$ обновляется в соответствии с положением i-й слева точки, формируя, таким образом, оболочку $\text{CH}(\{p_1, p_2, \ldots, p_i\})$. В упр. 33.3.6 предлагается показать, как реализовать этот метод так, чтобы время его работы было равно $O(n \lg n)$.
- В ***методе декомпозиции*** (devide-and-conquer method) множество n точек за время $\Theta(n)$ разбивается на два подмножества, в одном из которых содержится $\lceil n/2 \rceil$ крайних слева точек, а во втором — $\lfloor n/2 \rfloor$ крайних справа. Затем рекурсивно вычисляются выпуклые оболочки этих подмножеств, которые впоследствии объединяются с помощью одного остроумного метода за время $O(n)$. Время работы этого метода описывается знакомым рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, решение которого дает время работы $O(n \lg n)$.
- ***Метод отсечения и поиска*** (prune-and-search method) подобен алгоритму, описанному в разделе 9.3, время работы которого в наихудшем случае ведет себя линейно. В нем строится верхняя часть (или "верхняя цепь") выпуклой оболочки путем многократного отбрасывания фиксированной части оставшихся точек до тех пор, пока не останется только верхняя цепь выпуклой оболочки. Затем то же самое выполняется с нижней цепью. В асимптотическом пределе этот метод самый быстрый: если выпуклая оболочка содержит h вершин, время его работы равно $O(n \lg h)$.

Вычисление выпуклой оболочки множества точек — задача, интересная сама по себе. Кроме того, алгоритмы, предназначенные для решения некоторых других задач вычислительной геометрии, начинают работу с построения выпуклой оболочки. Например, рассмотрим двумерную задачу о ***поиске пары самых дальних точек*** (farthest-pair problem): в заданном на плоскости множестве n точек требуется найти две, расстояние между которыми является максимальным. В упр. 33.3.3 предлагается доказать, что обе эти точки должны быть вершинами

выпуклой оболочки. Хотя здесь мы и не станем доказывать это утверждение, пару самых дальних вершин выпуклого n-угольника можно найти в течение времени $O(n)$. Таким образом, путем построения выпуклой оболочки для n входных точек за время $O(n \lg n)$ и последующего поиска пары самых дальних точек среди вершин полученного в результате выпуклого многоугольника можно найти пару самых дальних точек из произвольного множества n точек за время $O(n \lg n)$.

Сканирование по Грэхему

В методе ***сканирования по Грэхему*** (Graham's scan) задача о выпуклой оболочке решается с помощью стека S, сформированного из точек-кандидатов. Все точки входного множества по одной Q заносятся в стек, а затем точки, не являющиеся вершинами $\text{CH}(Q)$, со временем удаляются из него. По завершении работы алгоритма в стеке S остаются только вершины оболочки $\text{CH}(Q)$ в порядке их обхода против часовой стрелки.

В качестве входных данных процедуры GRAHAM-SCAN выступает множество точек Q, где $|Q| \geq 3$. В ней вызывается функция TOP(S), которая возвращает точку, находящуюся на вершине стека S, не изменяя при этом его содержимое. Кроме того, используется также функция NEXT-TO-TOP(S), которая возвращает точку, расположенную в стеке S на одну позицию ниже от верхней точки; стек S при этом не изменяется. Вскоре будет показано, что стек S, возвращаемый процедурой GRAHAM-SCAN, содержит только вершины $\text{CH}(Q)$, причем расположенные в порядке обхода против часовой стрелки, если просматривать их в стеке снизу вверх.

```
GRAHAM-SCAN(Q)
Пусть p0 — точка Q с минимальной координатой y, или
    крайняя слева из таких точек при наличии совпадений
Пусть ⟨p1, p2, ..., pm⟩ – остальные точки Q, отсортированные
    в порядке возрастания полярного угла, измеряемого против
    часовой стрелки относительно p0 (если полярные углы
    нескольких точек совпадают, то из множества удаляются все
    эти точки, кроме одной, самой дальней от точки p0)
if m < 2
    return "выпуклая оболочка пуста"
else пусть S — пустой стек
    PUSH(p0, S)
    PUSH(p1, S)
    PUSH(p2, S)
    for i = 3 to m
        while угол, образованный точками NEXT-TO-TOP(S),
                TOP(S) и pi не образует поворот влево
            POP(S)
        PUSH(pi, S)
    return S
```

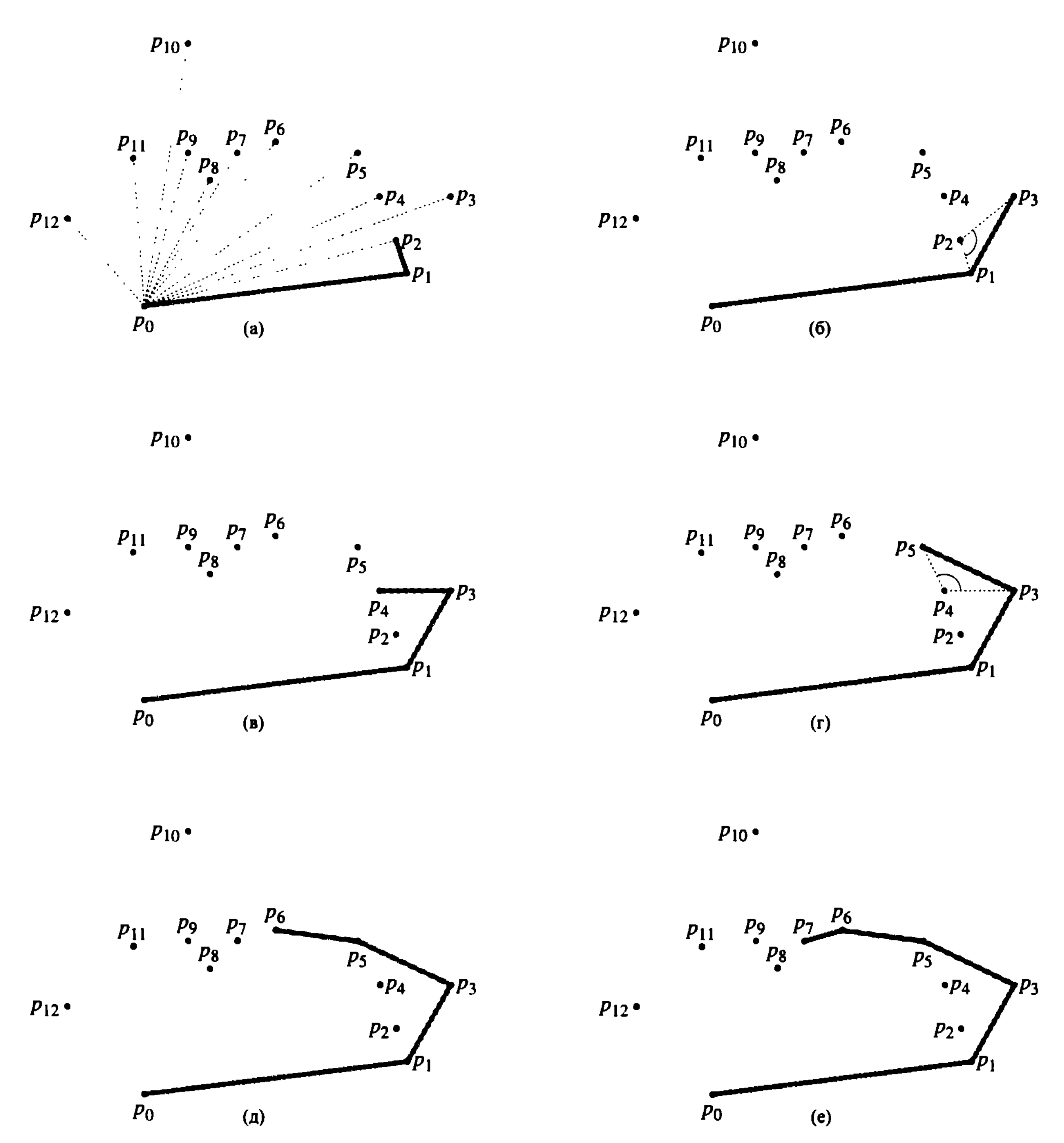


Рис. 33.7. Выполнение процедуры GRAHAM-SCAN для множества Q, показанного на рис. 33.6. Текущая выпуклая оболочка содержится в стеке S, показанном серым цветом на каждом шаге. **(а)** Последовательность $\langle p_1, p_2, \ldots, p_{12} \rangle$ точек, пронумерованных в порядке возрастания полярного угла относительно p_0, и исходный стек S, содержащий p_0, p_1 и p_2. **(б)–(л)** Стек S после каждой итерации цикла **for** в строках 9–12. Пунктирные линии показывают повороты не влево, которые приводят к снятию точек со стека. В части (з), например, поворот вправо в угле $\angle p_7 p_8 p_9$ приводит к снятию со стека p_8, а следующий затем поворот вправо в угле $\angle p_6 p_7 p_9$ приводит к снятию со стека точки p_7.

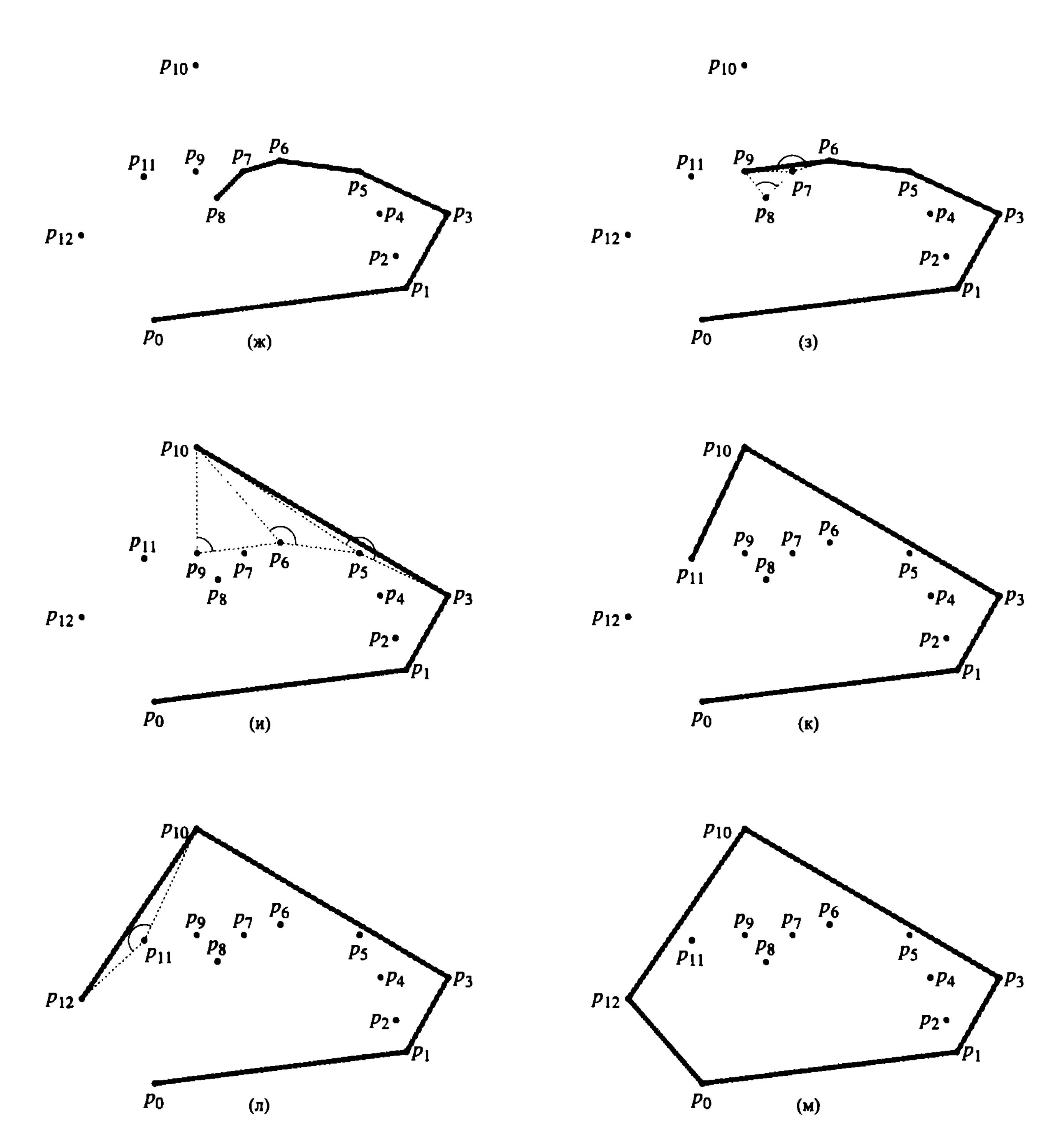


Рис. 33.7 (продолжение). **(м)** Выпуклая оболочка, возвращаемая процедурой и совпадающая с показанной на рис. 33.6.

Работа процедуры GRAHAM-SCAN проиллюстрирована на рис. 33.7. Выпуклая оболочка, содержащаяся на данный момент в стеке S, на каждом этапе показана серым цветом. В строке 1 процедуры выбирается точка p_0 с минимальной координатой y; при наличии нескольких таких точек выбирается крайняя слева. Поскольку в множестве Q отсутствуют точки, расположенные ниже точки p_0, а все другие точки с такой же координатой y расположены справа от точки p_0, точка p_0 должна быть вершиной оболочки $\mathrm{CH}(Q)$. В строке 2 остальные точки множества Q сортируются в порядке возрастания их полярных углов относительно точки p_0. При этом используется тот же метод, что и в упр. 33.1.3, т.е. сравнение векторных произведений. Если полярные углы двух или нескольких точек относительно точки p_0 совпадают, все такие точки, кроме самой удаленной от точки p_0, являются выпуклыми комбинациями точки p_0 и самой удаленной от нее из числа таких точек, поэтому все они могут быть исключены из рассмотрения. Обозначим через m количество оставшихся точек, отличных от точки p_0. Величина полярного угла каждой из точек множества Q, измеряемого в радианах относительно точки p_0, принадлежит полуоткрытому интервалу $[0, \pi)$. Поскольку точки отсортированы по величине возрастания их полярных углов, они расположены относительно точки p_0 в порядке обхода против часовой стрелки. Обозначим эту упорядоченную последовательность точек как $\langle p_1, p_2, \ldots, p_m \rangle$. Заметим, что точки p_1 и p_m являются вершинами оболочки $\mathrm{CH}(Q)$ (см. упр. 33.3.1). На рис. 33.7, (а) изображены точки из рис. 33.6, последовательно пронумерованные в порядке возрастания полярных углов относительно точки p_0.

В остальной части процедуры используется стек S. В строках 5–8 этот стек инициализируется, и в него заносятся снизу вверх три точки — p_0, p_1 и p_2. На рис. 33.7, (а) показано начальное состояние стека S. В каждой итерации цикла **for** в строках 9–12 обрабатывается очередная точка подпоследовательности $\langle p_3, p_4, \ldots, p_m \rangle$. Как мы увидим, в результате этой обработки точки p_i в стек S в порядке размещения снизу вверх оказываются помещенными вершины оболочки $\mathrm{CH}(\{p_0, p_1, \ldots, p_i\})$ в порядке против часовой стрелки. В цикле **while** в строках 10 и 11 точки удаляются из стека, если обнаруживается, что они не являются вершинами выпуклой оболочки. При обходе выпуклой оболочки против часовой стрелки в каждой вершине должен выполняться поворот влево. Каждый раз, когда в цикле **while** встречается вершина, в которой не происходит такой поворот влево, эта вершина снимается со стека. (Выполняется именно проверка того, что поворот происходит не влево, а не проверка того, что поворот происходит вправо. Такая проверка исключает наличие развернутого угла в полученной в результате выпуклой оболочке. Развернутых углов быть не должно, поскольку ни одна из вершин выпуклого многоугольника не может быть выпуклой комбинацией других вершин этого многоугольника.) После снятия со стека всех вершин, в которых при переходе к точке p_i не выполняется поворот влево, в стек заносится точка p_i. На рис. 33.7, (б)–(л) показано состояние стека S после каждой итерации цикла **for**. По окончании работы в строке 13 процедуры GRAHAM-SCAN возвращается стек S. На рис. 33.7, (м) показана выпуклая оболочка, полученная описанным способом.

В сформулированной ниже теореме формально доказывается корректность процедуры GRAHAM-SCAN.

Теорема 33.1 (Корректность сканирования по Грэхему)

Если процедура GRAHAM-SCAN выполняется над множеством точек Q, где $|Q| \geq 3$, то по завершении этой процедуры стек S будет содержать (в направлении снизу вверх) только вершины оболочки $\text{CH}(Q)$ в порядке обхода против часовой стрелки.

Доказательство. После выполнения строки 2 в нашем распоряжении оказывается последовательность точек $\langle p_1, p_2, \ldots, p_m \rangle$. Определим подмножество точек $Q_i = \{p_0, p_1, \ldots, p_i\}$ для $i = 2, 3, \ldots, m$. Множество точек $Q - Q_m$ образуют те из них, которые были удалены из-за того, что их полярный угол относительно точки p_0 совпадает с полярным углом некоторой точки из множества Q_m. Эти точки не принадлежат выпуклой оболочке $\text{CH}(Q)$, так что $\text{CH}(Q_m) = \text{CH}(Q)$. Таким образом, достаточно показать, что по завершении процедуры GRAHAM-SCAN стек S состоит из вершин оболочки $\text{CH}(Q_m)$ в порядке обхода против часовой стрелки, если перечислять эти точки в стеке снизу вверх. Заметим, что точно так же, как точки p_0, p_1 и p_m являются вершинами оболочки $\text{CH}(Q)$, точки p_0, p_1 и p_i являются вершинами оболочки $\text{CH}(Q_i)$.

В доказательстве используется следующий инвариант цикла.

> В начале каждой итерации цикла **for** в строках 9–12 стек S состоит (снизу вверх) только из вершин оболочки $\text{CH}(Q_{i-1})$ в порядке их обхода против часовой стрелки.

Инициализация. При первом выполнении строки 9 инвариант выполняется, поскольку в этот момент стек S состоит только из вершин $Q_2 = Q_{i-1}$, и это множество трех вершин формирует собственную выпуклую оболочку. Кроме того, если просматривать точки снизу вверх, то они будут расположены в порядке обхода против часовой стрелки.

Сохранение. При входе в новую итерацию цикла **for** вверху стека S находится точка p_{i-1}, помещенная туда в конце предыдущей итерации (или перед первой итерацией, когда $i = 3$). Пусть p_j — верхняя точка стека S после выполнения цикла **while** в строках 10 и 11, но до того, как в строке 12 в стек будет помещена точка p_i. Пусть также p_k — точка, расположенная в стеке S непосредственно под точкой p_j. В тот момент, когда точка p_j является верхней точкой стека S, а точка p_i еще туда не добавлена, стек содержит те же точки, что и после j-й итерации цикла **for**. Поэтому согласно инварианту цикла в этот момент стек S содержит только вершины $\text{CH}(Q_j)$, и они располагаются в порядке их обхода против часовой стрелки, если просматривать их снизу вверх.

Продолжим рассматривать момент непосредственно перед внесением в стек точки p_i. Мы знаем, что полярный угол точки p_i относительно точки p_0 больше, чем полярный угол точки p_j, и что угол $\angle p_k p_j p_i$ сворачивает влево (в противном случае точка p_j была бы снята со стека). Следовательно, поскольку S содержит в точности вершины $\text{CH}(Q_j)$, из рис. 33.8, (а) мы видим, что после

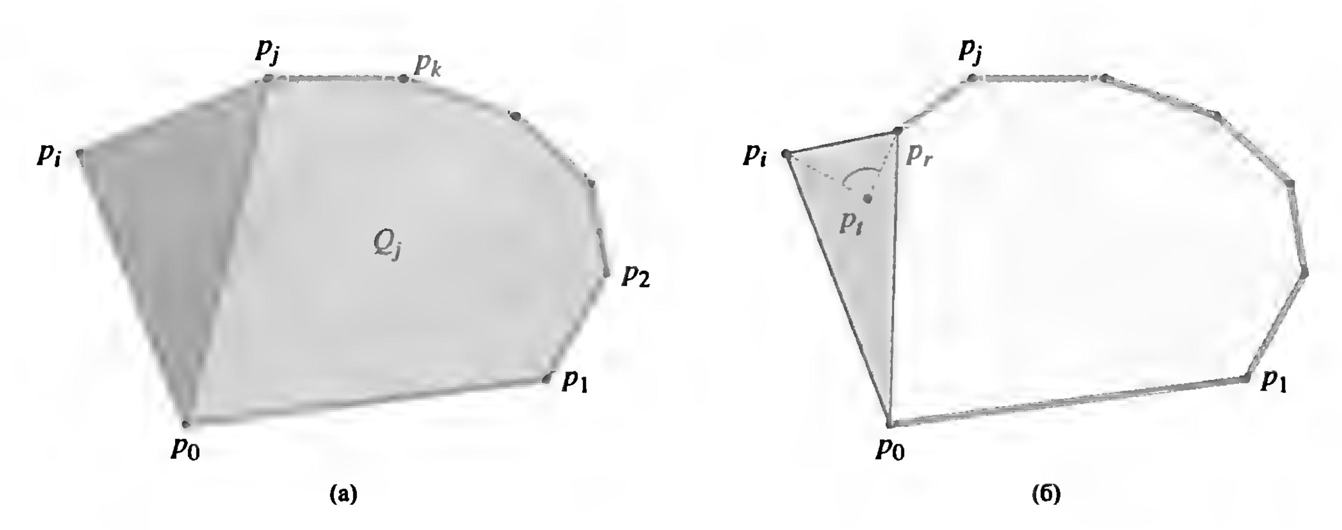


Рис. 33.8. Доказательство корректности процедуры GRAHAM-SCAN. **(а)** Поскольку полярный угол p_i относительно p_0 больше полярного угла p_j и поскольку угол $\angle p_k p_j p_i$ представляет собой поворот влево, добавление p_i к $\text{CH}(Q_j)$ в точности дает вершины $\text{CH}(Q_j \cup \{p_i\})$. **(б)** Если угол $\angle p_r p_t p_i$ не делает поворот влево, то p_t либо представляет собой внутреннюю точку треугольника, образованного точками p_0, p_r и p_i, либо находится на стороне треугольника, а это означает, что она не может быть вершиной $\text{CH}(Q_i)$.

внесения p_i содержимое стека S будет представлять собой в точности вершины $\text{CH}(Q_j \cup \{p_i\})$, находящиеся в том же порядке против часовой стрелки при перечислении их снизу вверх.

Теперь покажем, что множество вершин $\text{CH}(Q_j \cup \{p_i\})$ совпадает с множеством точек $\text{CH}(Q_i)$. Рассмотрим произвольную точку p_t, снятую со стека во время выполнения i-й итерации цикла **for**, и пусть p_r — точка, расположенная в стеке S непосредственно под точкой p_t перед снятием со стека последней (этой точкой p_r может быть точка p_j). Угол $\angle p_r p_t p_i$ не сворачивает влево, а полярный угол точки p_t относительно точки p_0 больше полярного угла точки p_r. Как видно из рис. 33.8, (б), точка p_t должна располагаться либо внутри треугольника, образованного точками p_0, p_r и p_i, либо на стороне этого треугольника (но не совпадать с его вершиной). Очевидно, что поскольку точка p_t находится внутри треугольника, образованного тремя другими точками множества Q_i, она не может быть вершиной $\text{CH}(Q_i)$. Так как p_t не является вершиной $\text{CH}(Q_i)$, можно записать соотношение

$$\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i) \ . \qquad (33.1)$$

Пусть P_i является множеством точек, снятых со стека во время выполнения i-й итерации цикла **for**. Поскольку равенство (33.1) применимо ко всем точкам множества P_i, в результате его многократного применения можно показать, что выполняется равенство $\text{CH}(Q_i - P_i) = \text{CH}(Q_i)$. Однако $Q_i - P_i = Q_j \cup \{p_i\}$, поэтому мы приходим к заключению, что $\text{CH}(Q_j \cup \{p_i\}) = \text{CH}(Q_i - P_i) = \text{CH}(Q_i)$.

Мы показали, что сразу после снятия со стека S точки p_i в нем содержатся только все вершины $\text{CH}(Q_i)$ в порядке их обхода против часовой стрелки, если просматривать их в стеке снизу вверх. Последующее увеличение на единицу

значения переменной i приведет к сохранению инварианта цикла в очередной итерации.

Завершение. По завершении цикла выполняется равенство $i = m+1$, поэтому из инварианта цикла следует, что стек S состоит в точности из вершин $\text{CH}(Q_m)$, т.е. из вершин $\text{CH}(Q)$, в порядке обхода против часовой стрелки, если они просматриваются в стеке снизу вверх. На этом доказательство завершается. ■

Теперь покажем, что время работы процедуры GRAHAM-SCAN равно $O(n \lg n)$, где $n = |Q|$. Выполнение строки 1 занимает время $\Theta(n)$. Для сортировки полярных углов методом слияния или пирамидальной сортировки, а также для сравнения углов с помощью векторного произведения, как описано в разделе 33.1, в строке 2 требуется время $O(n \lg n)$. (Все точки с одним и тем же значением полярного угла, кроме самой удаленной, можно удалить за время $O(n)$.) Выполнение строк 5–8 занимает время $O(1)$. В силу неравенства $m \leq n - 1$ цикл **for** в строках 9–12 выполняется не более $n - 3$ раз. Так как выполнение процедуры PUSH занимает время $O(1)$, на выполнение каждой итерации требуется время $O(1)$, если не принимать во внимание время, затраченное на выполнение цикла **while** в строках 10 и 11. Таким образом, общее время выполнения цикла **for** равно $O(n)$, за исключением времени выполнения вложенного в него цикла **while**.

Покажем с помощью группового анализа, что для выполнения всего цикла **while** потребуется время $O(n)$. При $i = 0, 1, \ldots, m$ каждая точка p_i попадает в стек S ровно по одному разу. Как и в ходе анализа процедуры MULTIPOP в разделе 17.1, нетрудно заметить, что каждой операции PUSH соответствует не более одной операции POP. По крайней мере три точки (p_0, p_1 и p_m) никогда не снимаются со стека, поэтому всего выполняется не более $m-2$ операций POP. В каждой итерации цикла **while** выполняется одна операция POP, следовательно, всего этих итераций не более $m - 2$. Так как проверка в строке 10 занимает время $O(1)$, каждый вызов POP требует времени $O(1)$, и $m \leq n - 1$, полное время, которое требуется для выполнения цикла **while**, равно $O(n)$. Таким образом, время выполнения процедуры GRAHAM-SCAN составляет $O(n \lg n)$.

Обход по Джарвису

При построении выпуклой оболочки множества точек Q путем ***обхода по Джарвису*** (Jarvis's march) используется метод, известный как ***упаковка пакета*** (package wrapping) или ***упаковка подарка*** (gift wrapping). Время выполнения алгоритма равно $O(nh)$, где h – количество вершин $\text{CH}(Q)$. В том случае, когда h равно $o(\lg n)$, обход по Джарвису работает быстрее, чем сканирование по Грэхему.

Интуитивно обход по Джарвису моделирует обертывание плотного куска бумаги вокруг множества Q. Начнем с того, что прикрепим конец бумаги к самой низкой точке множества, т.е. к той же точке p_0, с которой начинается сканирование по Грэхему. Эта точка является вершиной выпуклой оболочки. Натянем бумагу вправо, чтобы она не провисала, после чего будем перемещать ее вверх до тех пор, пока она не коснется какой-либо точки. Эта точка также должна быть

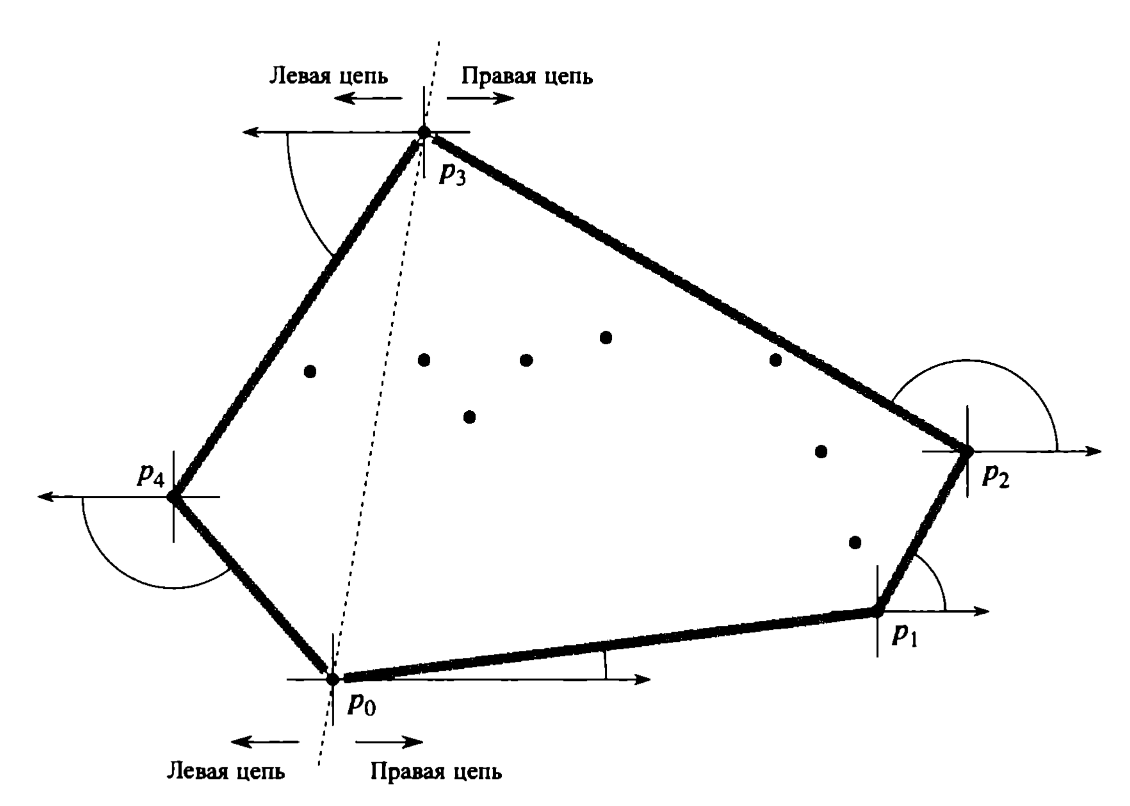


Рис. 33.9. Обход по Джарвису. В качестве первой вершины выбирается наинизшая точка p_0. Следующая вершина, p_1, имеет наименьший полярный угол по отношению к p_0 среди всех точек. Затем точка p_2 имеет наименьший полярный угол по отношению к p_1. Правая цепь идет вверх до самой верхней точки p_3. Затем выполняется построение левой цепи путем поиска наименьших полярных углов относительно отрицательного направления оси x.

вершиной выпуклой оболочки. Сохраняя бумагу натянутой, продолжим наматывать ее на множество вершин, пока не вернемся к исходной точке p_0.

Если говорить более формально, то при обходе по Джарвису строится последовательность $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ вершин $\mathrm{CH}(Q)$. Начнем с точки p_0. Как видно из рис. 33.9, следующая вершина выпуклой оболочки p_1 имеет наименьший полярный угол относительно точки p_0. (При наличии совпадений выбирается точка, наиболее удаленная от точки p_0.) Аналогично точка p_2 имеет наименьший полярный угол относительно точки p_1 и т.д. По достижении самой высокой вершины, скажем, p_k (совпадения разрешаются путем выбора самой удаленной из таких вершин), оказывается построенной (рис. 33.9) ***правая цепь*** (right chain) оболочки $\mathrm{CH}(Q)$. Чтобы сконструировать ***левую цепь*** (left chain), начнем с точки p_k и выберем в качестве p_{k+1} точку с наименьшим полярным углом относительно точки p_k, но *относительно отрицательного направления оси x*. Продолжаем выполнять эту процедуру, отсчитывая полярный угол от отрицательного направления оси x, пока не вернемся к исходной точке p_0.

Обход по Джарвису можно было бы реализовать путем единообразного обхода вокруг выпуклой оболочки, т.е. не прибегая к отдельному построению правой и левой цепей. В такой реализации обычно отслеживается угол последней стороны выпуклой оболочки и накладывается требование, чтобы последовательность углов, образованных сторонами оболочки с положительным направлением оси x, строго возрастала (в интервале от 0 до 2π радиан). Преимущество конструирования отдельных цепей заключается в том, что исключается необходимость

явно вычислять углы; достаточно сравнения углов методами, описанными в разделе 33.1.

При надлежащей реализации время выполнения обхода по Джарвису равно $O(nh)$. Для каждой из h вершин оболочки $\text{CH}(Q)$ выполняется поиск вершины с минимальным полярным углом. Каждое сравнение полярных углов выполняется за время $O(1)$, если использовать методы, описанные в разделе 33.1. Как показано в разделе 9.1, если каждая операция сравнения выполняется за время $O(1)$, то выбор минимального из n значений занимает время $O(n)$. Таким образом, обход по Джарвису завершается за время $O(nh)$.

Упражнения

33.3.1

Докажите, что в процедуре GRAHAM-SCAN точки p_1 и p_m должны быть вершинами $\text{CH}(Q)$.

33.3.2

Рассмотрим модель вычислений, которая поддерживает сложение, сравнение и умножение и в которой существует нижняя граница времени сортировки n чисел, равная $\Omega(n \lg n)$. Докажите, что в такой модели $\Omega(n \lg n)$ — нижняя граница времени вычисления вершин выпуклой оболочки множества n точек в порядке их обхода.

33.3.3

Докажите, что в заданном множестве точек Q две самые удаленные одна от другой точки должны быть вершинами $\text{CH}(Q)$.

33.3.4

Для заданного многоугольника P и точки q на его границе ***тенью*** (shadow) точки q называется множество точек r, для которых отрезок $\overline{qr}$ полностью находится на границе или во внутренней области многоугольника P. Как показано на рис. 33.10, многоугольник P является ***звездообразным*** (star-shaped), если в его внутренней области существует точка p, которая находится в тени любой точки, лежащей на границе этого многоугольника. Множество всех таких точек называется ***ядром*** (kernel) многоугольника P. Пусть звездообразный n-угольник P задан своими вершинами, перечисленными в порядке обхода против часовой стрелки. Покажите, как построить $\text{CH}(P)$ за время $O(n)$.

33.3.5

В ***оперативной задаче о выпуклой оболочке*** (on-line convex-hull problem) параметры каждой из n точек из заданного множества Q становятся известными по одной точке за раз. После получения сведений об очередной точке строится выпуклая оболочка тех точек, о которых стало известно к настоящему времени. Очевидно, можно было бы применять сканирование по Грэхему к каждой из точек, затратив при этом полное время, равное $O(n^2 \lg n)$. Покажите, как решить оперативную задачу о выпуклой оболочке за время $O(n^2)$.

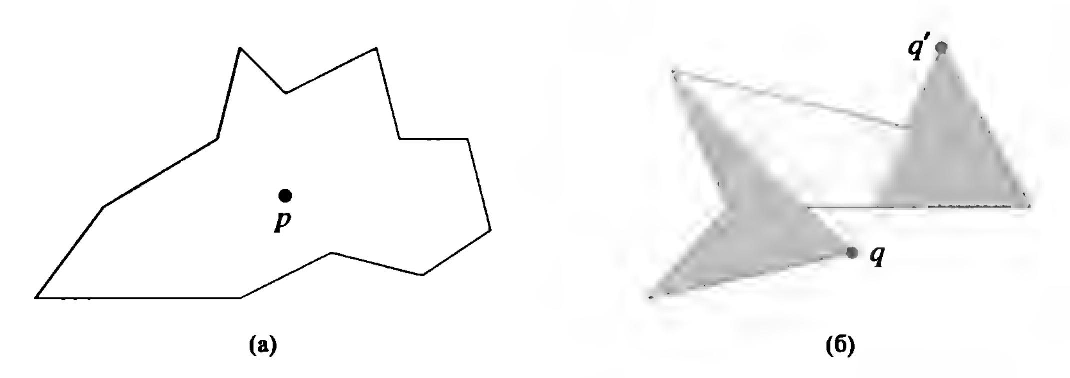


Рис. 33.10. Определение звездообразного многоугольника, о котором идет речь в упр. 33.3.4. **(а)** Звездообразный многоугольник. Отрезок, соединяющий точку p с любой из принадлежащих границе многоугольника точек q, пересекает эту границу только в точке q. **(б)** Многоугольник, не являющийся звездообразным. Выделенная серым цветом левая область является тенью точки q, а выделенная правая область — тенью точки q'. Поскольку эти области не перекрываются, ядро является пустым.

33.3.6 ★

Покажите, как реализовать инкрементный метод построения выпуклой оболочки n точек таким образом, чтобы время его работы было равно $O(n \lg n)$.

33.4. Поиск пары ближайших точек

Теперь рассмотрим задачу о поиске в множестве Q, состоящем из $n \geq 2$ точек, пары точек, ближайших одна к другой. Термин "ближайший" относится к обычному евклидову расстоянию: расстояние между точками $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ равно $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Две точки множества Q могут совпадать; в этом случае расстояние между ними равно нулю. Эта задача находит применение, например, в системах управления движением транспорта. В системе управления воздушным или морским транспортом может понадобиться узнать, какие из двух транспортных средств находятся друг к другу ближе всего, чтобы предотвратить возможное столкновение между ними.

В алгоритме, работающем "в лоб", просто перебираются все $\binom{n}{2} = \Theta(n^2)$ пар точек. В данном разделе описывается решение этой задачи с помощью алгоритма разбиения ("разделяй и властвуй"). Время решения этим методом определяется знакомым рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$. Таким образом, время работы этого алгоритма равно $O(n \lg n)$.

Алгоритм декомпозиции

В каждом рекурсивном вызове описываемого алгоритма в качестве входных данных используются подмножество $P \subseteq Q$ и массивы X и Y, в каждом из которых содержатся все точки входного подмножества P. Точки в массиве X отсортированы в порядке возрастания их координаты x. Аналогично массив Y отсортирован в порядке возрастания координаты y. Заметим, что достигнуть границы $O(n \lg n)$ не удастся, если сортировка будет выполняться в каждом рекурсивном

вызове; если бы мы поступали таким образом, то время работы такого метода определялось бы рекуррентным соотношением $T(n) = 2T(n/2) + O(n \lg n)$, решение которого равно $T(n) = O(n \lg^2 n)$. (Это можно показать с помощью версии основного метода, описанной в упр. 4.6.2.) Немного позже станет понятно, как с помощью "предварительной сортировки" поддерживать свойство отсортированности, не прибегая к процедуре сортировки в каждом рекурсивном вызове.

В рекурсивном вызове со входными данными P, X и Y сначала проверяется, выполняется ли условие $|P| \leq 3$. Если оно справедливо, то в вызове просто применяется упомянутый выше метод решения "в лоб": сравниваются между собой все $\binom{|P|}{2}$ пар точек и возвращается пара точек, расположенных друг к другу ближе других. Если же $|P| > 3$, то выполняется описанный ниже рекурсивный вызов в соответствии с парадигмой "разделяй и властвуй".

Разделение. Выполняется поиск вертикальной прямой l, которая делит множество точек P на два множества P_L и P_R, такие, что $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, все точки множества P_L находятся слева от прямой l или на этой прямой, а все точки множества P_R находятся справа от прямой l или на этой прямой. Массив X разбивается на массивы X_L и X_R, содержащие точки множеств P_L и P_R соответственно, отсортированные в порядке возрастания координаты x. Аналогично массив Y разбивается на массивы Y_L и Y_R, содержащие соответственно точки множеств P_L и P_R, отсортированные в порядке монотонного возрастания координаты y.

Властвование. После разбиения множества P на подмножества P_L и P_R выполняются два рекурсивных вызова: один — для поиска пары ближайших точек в множестве P_L, а другой — для поиска пары ближайших точек в множестве P_R. В качестве входных данных в первом вызове выступает подмножество P_L и массивы X_L и Y_L; во втором вызове на вход подаются множество P_R, а также массивы X_R и Y_R. Обозначим расстояния между ближайшими точками в множествах P_L и P_R как δ_L и δ_R соответственно; введем также обозначение $\delta = \min(\delta_L, \delta_R)$.

Комбинирование. Ближайшая пара либо находится друг от друга на расстоянии δ, найденном в одном из рекурсивных вызовов, либо образована точками, одна из которых принадлежит множеству P_L, а вторая — множеству P_R. В алгоритме определяется, существует ли пара таких точек, расстояние между которыми меньше δ. Заметим, что если существует такая "пограничная" пара точек, расстояние между которыми меньше δ, то обе они не могут находиться от прямой l дальше, чем на расстоянии δ. Таким образом, как видно из рис. 33.11, (а), обе эти точки должны лежать в пределах вертикальной полосы шириной 2δ, в центре которой находится прямая l. Для поиска такой пары (если она существует) в алгоритме выполняются описанные ниже действия.

1. Создается массив Y' путем удаления из массива Y всех точек, не попадающих в полосу шириной 2δ. Как и в массиве Y, в массиве Y' точки отсортированы в порядке возрастания координаты y.

2. Для каждой точки p из массива Y' алгоритм пытается найти в этом же массиве точки, которые находятся в δ-окрестности точки p. Как мы вскоре увидим, достаточно рассмотреть лишь 7 точек, расположенных в массиве Y' после точки p. В алгоритме вычисляется расстояние от точки p до каждой из этих семи точек, и отслеживается наименьшее расстояние между точками δ', вычисленное по всем парам точек в массиве Y'.
3. Если $\delta' < \delta$, то в вертикальной полосе действительно содержится пара точек, которые находятся одна к другой ближе, чем те, которые были найдены в результате рекурсивных вызовов. В таком случае возвращаются эта пара точек и соответствующее ей расстояние δ'. В противном случае возвращаются пара ближайших точек и расстояние между ними δ, найденные в результате рекурсивных вызовов.

В приведенном выше описании опущены некоторые детали реализации, необходимые для сокращения времени работы до величины $O(n \lg n)$. После доказательства корректности алгоритма будет показано, как реализовать этот алгоритм так, чтобы достичь желаемой границы времени работы.

Корректность

Корректность описанного алгоритма поиска пары ближайших точек очевидна, за исключением двух аспектов. Во-первых, ограничивая рекурсивный спуск условием $|P| \leq 3$, мы гарантируем, что никогда не придется решать вспомогательную задачу, в которой задана только одна точка. Второй аспект заключается в том, что требуется проверить всего 7 точек, расположенных в массиве Y' за каждой точкой p; далее будет приведено доказательство этого свойства.

Предположим, что на некотором уровне рекурсии пару ближайших точек образуют точки $p_L \in P_L$ и $p_R \in P_R$. Таким образом, расстояние δ' между этими точками строго меньше δ. Точка p_L должна находиться слева от прямой l на расстоянии, не превышающем величину δ, или на самой этой прямой. Аналогично точка p_R находится справа от прямой l на расстоянии, не превышающем величину δ, или на самой этой прямой. Кроме того, расстояние по вертикали между точками p_L и p_R не превышает δ. Таким образом, как видно из рис. 33.11, (а), точки p_L и p_R находятся внутри прямоугольника $\delta \times 2\delta$, через центр которого проходит прямая l. (В этот прямоугольник могут попасть и другие точки.)

Теперь покажем, что в прямоугольнике размером $\delta \times 2\delta$ может содержаться не более восьми точек из P. Рассмотрим квадрат $\delta \times \delta$, который составляет левую половину этого прямоугольника. Поскольку расстояние между любой парой точек из множества P_L не меньше δ, в этом квадрате может находиться не более четырех точек (рис. 33.11, (б)). Аналогично в квадрате размером $\delta \times \delta$, составляющем правую половину указанного прямоугольника, также может быть не более четырех точек. Таким образом, в прямоугольнике $\delta \times 2\delta$ содержится не более восьми точек. (Заметим, что, поскольку точки на прямой l могут входить либо в множество P_L, либо в множество P_R, на этой прямой может лежать до четырех точек. Этот предел достигается, если имеется две пары совпадающих точек, причем в каждой паре одна из точек принадлежит множеству P_L, а другая — множеству

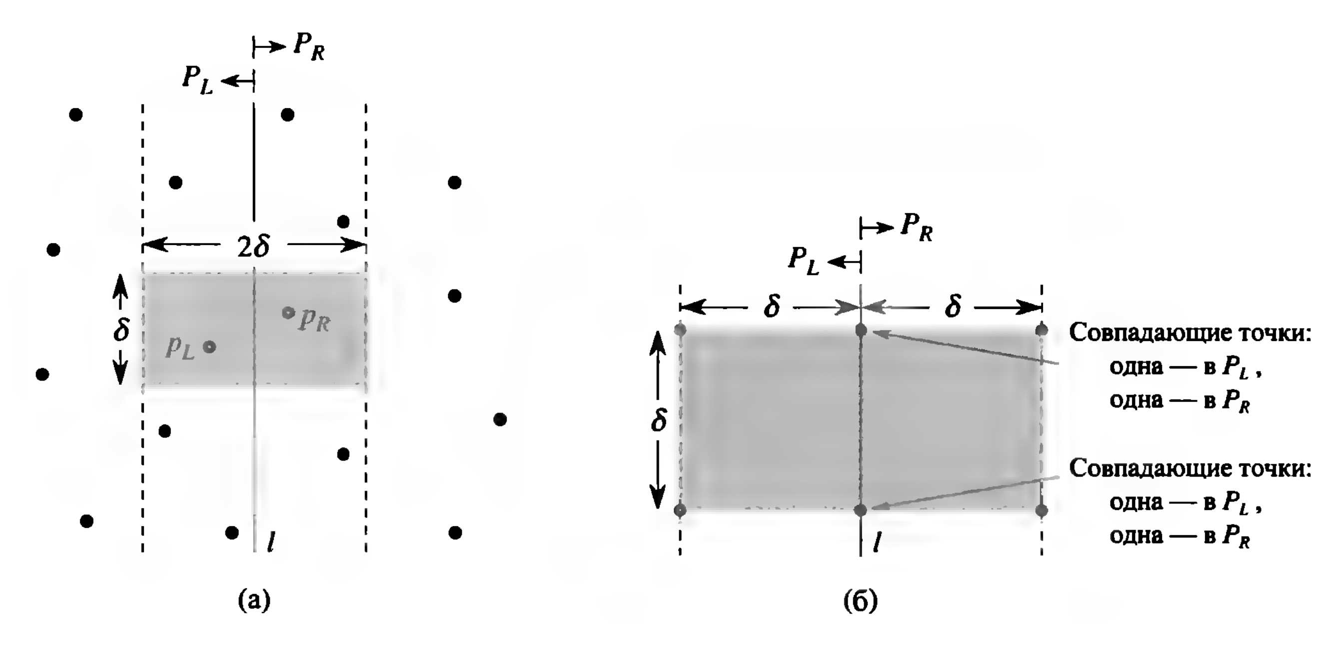


Рис. 33.11. Ключевые концепции, используемые в доказательстве того, что алгоритму поиска пары ближайших точек достаточно проверить только семь точек, следующих за каждой точкой в массиве Y'. **(а)** Если точки $p_L \in P_L$ и $p_R \in P_R$ находятся на расстоянии меньше δ одна от другой, то они должны располагаться в пределах прямоугольника $\delta \times 2\delta$, посредине которого проходит прямая l. **(б)** Как четыре точки, попарно расположенные на расстоянии не менее δ одна от другой, могут располагаться в пределах квадрата $\delta \times \delta$. Слева показаны четыре точки из P_L, а справа — четыре точки из P_R. Прямоугольник $\delta \times 2\delta$ может содержать восемь точек, если на самом деле точки, показанные на прямой l, представляют собой пары совпадающих точек, из которых одна находится в P_L, а другая — в P_R.

P_R. Кроме того, одна из этих пар находится на пересечении прямой l и верхней стороны прямоугольника, а другая — на пересечении прямой l и нижней стороны прямоугольника.)

Поскольку в прямоугольнике указанных выше размеров может содержаться не более восьми точек множества P, очевидно, что достаточно проверить семь точек, расположенных в массиве Y' после каждой точки. Продолжая придерживаться предположения о том, что пара ближайших одна к другой точек состоит из точек p_L и p_R, без потери общности предположим, что в массиве Y' точка p_L находится перед точкой p_R. В этом случае точка p_R является одной из семи точек, следующих после точки p_L, даже если точка p_L встречается в массиве Y' настолько рано, насколько это возможно, а точка p_R — настолько поздно, насколько это возможно. Таким образом, корректность алгоритма, предназначенного для поиска пары ближайших точек, доказана.

Реализация и время работы алгоритма

Как уже упоминалось, наша цель — сделать так, чтобы время работы представленного алгоритма описывалось рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, где $T(n)$ — время работы алгоритма для n точек. Основная сложность — это добиться, чтобы массивы X_L, X_R, Y_L и Y_R, которые передаются в рекурсивных вызовах, были отсортированы по соответствующей координате и чтобы массив Y' был отсортирован по координате y. (Заметим, что если массив X, который

передается в рекурсивном вызове, уже отсортирован, то разбиение множества P на подмножества P_L и P_R легко выполнить за время, линейно зависящее от количества элементов.)

Ключевое наблюдение состоит в том, что в каждом вызове следует сформировать отсортированное подмножество отсортированного массива. Например, пусть в каком-то отдельно взятом рекурсивном вызове заданы подмножество P и массив Y, отсортированный по координате y. В процессе разбиения множества P на подмножества P_L и P_R нужно образовать массивы Y_L и Y_R, которые должны быть отсортированы по координате y, причем эти массивы необходимо сформировать в течение линейного времени. Этот метод можно рассматривать как антипод процедуры MERGE, описанной в разделе 2.3.1: отсортированный массив разбивается на два отсортированных массива. Эта идея реализована в приведенном ниже псевдокоде.

```
Пусть Y_L[1..Y.length] и Y_R[1..Y.length] — новые массивы
Y_L.length = Y_R.length = 0
for i = 1 to Y.length
    if Y[i] ∈ P_L
        Y_L.length = Y_L.length + 1
        Y_L[Y_L.length] = Y[i]
    else Y_R.length = Y_R.length + 1
        Y_R[Y_R.length] = Y[i]
```

Мы просто проверяем точки массива Y в порядке их расположения в этом массиве. Если точка $Y[i]$ принадлежит множеству P_L, она добавляется в конец массива Y_L; в противном случае она добавляется в конец массива Y_R. С помощью аналогичного псевдокода можно сформировать массивы X_L, X_R и Y'.

Последний вопрос — как сделать так, чтобы точки были отсортированы с самого начала. Для этого следует выполнить их ***предварительную сортировку*** (presorting), которая осуществляется один раз *перед* первым рекурсивным вызовом. Отсортированные массивы передаются в первом рекурсивном вызове; затем они будут дробиться в рекурсивных вызовах до тех пор, пока это будет необходимо. Предварительная сортировка добавит ко времени работы алгоритма величину $O(n \lg n)$, но позволит выполнять каждый шаг рекурсии в течение линейного времени. Таким образом, если обозначить через $T(n)$ время обработки каждого шага рекурсии, а через $T'(n)$ — общее время работы всего алгоритма, то можно получить равенство $T'(n) = T(n) + O(n \lg n)$ и

$$T(n) = \begin{cases} 2T(n/2) + O(n) \;, & \text{если } n > 3 \;, \\ O(1) \;, & \text{если } n \leq 3 \;. \end{cases}$$

Таким образом, $T(n) = O(n \lg n)$ и $T'(n) = O(n \lg n)$.

Упражнения

33.4.1

Профессор предложил схему, позволяющую ограничиться в алгоритме поиска пары ближайших точек пятью точками, которые находятся в массиве Y' после каждой из точек. Его идея заключается в том, чтобы точки, которые лежат на прямой l, всегда заносились в множество P_L. Тогда на этой прямой не может быть пар совпадающих точек, одна из которых принадлежит множеству P_L, а другая — множеству P_R. Таким образом, прямоугольник размером $\delta \times 2\delta$ может содержать не более шести точек. Какая ошибка содержится в предложенной профессором схеме?

33.4.2

Покажите, что в действительности достаточно проверки точек только в пяти позициях массива, следующих за каждой точкой в массиве Y'.

33.4.3

Расстояние между двумя точками можно определить и не в евклидовом смысле, а по-другому. Так, ***L_m-расстояние*** (L_m-distance) между точками p_1 и p_2 на плоскости задается выражением $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Таким образом, евклидово расстояние — это L_2-расстояние. Модифицируйте алгоритм поиска пары ближайших точек для L_1-расстояния, известного как ***манхэттенское расстояние*** (Manhattan distance).

33.4.4

Для двух точек, p_1 и p_2, заданных на плоскости, L_∞-расстоянием между ними называется величина $\max(|x_1 - x_2|, |y_1 - y_2|)$. Модифицируйте алгоритм поиска пары ближайших точек для L_∞-расстояния.

33.4.5

Предположим, что среди переданных алгоритму поиска пар ближайших точек имеется $\Omega(n)$ ковертикальных. Покажите, как определить множества P_L и P_R и как определить, находится ли каждая точка Y в P_L или P_R так, чтобы время работы алгоритма осталось равным $O(n \lg n)$.

33.4.6

Предложите, как изменить алгоритм поиска пары ближайших точек, чтобы избежать в нем предварительной сортировки массива Y, но чтобы время его работы по-прежнему оставалось равным $O(n \lg n)$. (*Указание:* объедините отсортированные массивы Y_L и Y_R в отсортированный массив Y.)

Задачи

33.1. Выпуклые слои

Для заданного на плоскости множества точек Q ***выпуклые слои*** (convex layers) определяются следующим индуктивным образом. Первый выпуклый слой множества Q состоит из вершин $\text{CH}(Q)$. Определим для $i > 1$ множество Q_i, состоящее из точек множества Q, из которого удалили все точки выпуклых слоев $1, 2, \ldots, i-1$. Тогда i-м выпуклым слоем множества Q является $\text{CH}(Q_i)$, если $Q_i \neq \emptyset$; в противном случае он считается неопределенным.

а. Разработайте алгоритм, который находил бы выпуклые слои множества из n точек за время $O(n^2)$.

б. Докажите, что для построения выпуклых слоев в множестве из n точек по любой вычислительной модели, в которой сортировка n действительных чисел занимает время $\Omega(n \lg n)$, требуется время $\Omega(n \lg n)$.

33.2. Максимальные слои

Пусть Q — множество n точек на плоскости. Говорят, что точка (x, y) ***доминирует*** (dominates) над точкой (x', y'), если выполняются неравенства $x \geq x'$ и $y \geq y'$. Точка множества Q, над которой не доминирует никакая другая точка этого множества, называется ***максимальной*** (maximal). Заметим, что множество Q может содержать большое количество максимальных точек, которые можно объединить в ***максимальные слои*** (maximal layers) следующим образом. Первый максимальный слой L_1 представляет собой множество максимальных точек множества Q. Для $i > 1$ определим i-й максимальный слой L_i как множество максимальных точек в $Q - \bigcup_{j=1}^{i-1} L_j$.

Предположим, что множество Q содержит k непустых максимальных слоев, и пусть y_i — координата y крайней слева точки в слое L_i ($i = 1, 2, \ldots, k$). Мы предполагаем, что координаты x или y никаких двух точек множества Q не совпадают.

а. Покажите, что $y_1 > y_2 > \cdots > y_k$.

Рассмотрим точку (x, y), которая находится левее всех точек множества Q и координата y которой отличается от координаты y любой другой точки этого множества. Введем обозначение $Q' = Q \cup \{(x, y)\}$.

б. Пусть j — минимальный индекс, такой что $y_j < y$; если такого индекса нет ($y < y_k$), то $j = k + 1$. Покажите, что максимальные слои множества Q' обладают следующими свойствами.

- Если $j \leq k$, то максимальные слои множества Q' совпадают с максимальными слоями множества Q, за исключением слоя L_j, который в качестве новой крайней слева точки включает точку (x, y).

- Если $j = k + 1$, то первые k максимальных слоев множества Q' совпадают с максимальными слоями множества Q, но, кроме того, множество Q' имеет непустой $(k+1)$-й максимальный слой $L_{k+1} = \{(x, y)\}$.

в. Разработайте алгоритм, позволяющий за время $O(n \lg n)$ построить максимальные слои множества Q, состоящего из n точек. (*Указание:* проведите вертикальную выметающую прямую справа налево.)

г. Возникнут ли какие-либо сложности, если разрешить входным точкам иметь одинаковые координаты x или y? Предложите способ решения таких проблем.

33.3. Привидения и охотники за ними

Группа, состоящая из n охотников за привидениями, борется с n привидениями. Каждый охотник вооружен протонной установкой, уничтожающей привидение протонным пучком. Поток протонов распространяется по прямой и прекращает свой путь, когда попадает в привидение. Охотники придерживаются следующей стратегии. Каждый из них выбирает среди привидений свою цель, в результате чего образуется n пар "охотник–привидение". После этого каждый охотник выпускает пучок протонов по своей жертве. Известно, что пересечение пучков *очень* опасно, поэтому охотники должны выбирать привидения так, чтобы избежать таких пересечений.

Предположим, что положение каждого охотника и каждого привидения задано фиксированной точкой на плоскости и что никакие три точки не коллинеарны.

а. Докажите, что всегда существует прямая, проходящая через одного охотника и одно привидение, для которой количество охотников, попавших в одну из полуплоскостей относительно этой прямой, равно количеству привидений в этой же полуплоскости. Опишите, как найти такую прямую за время $O(n \lg n)$.

б. Разработайте алгоритм, позволяющий в течение времени $O(n^2 \lg n)$ разбить охотников и привидения на пары таким образом, чтобы не было пересечения пучков.

33.4. Сбор палочек

Профессор[3] занимается исследованием множества из n палочек, сложенных в кучу в некоторой конфигурации. Положение каждой палочки задается ее конечными точками, а каждая конечная точка представляет собой упорядоченную тройку координат (x, y, z). Вертикальных палочек нет. Профессор хочет собрать по одной все палочки, соблюдая условие, согласно которому он может снимать палочку только тогда, когда поверх нее не лежат никакие другие палочки.

---

[3] В оригинале — непереводимая игра слов: профессор Харон и палочки (sticks). Харон в греческой мифологии — перевозчик душ умерших через реку Стикс в Аид (подземное царство мертвых). — *Примеч. пер.*

а. Разработайте процедуру, которая для двух заданных палочек, a и b, определяла бы, находится ли палочка a над палочкой b, под ней или палочка a не связана с палочкой b ни одним из этих соотношений.

б. Разработайте эффективный алгоритм, позволяющий определить, можно ли собрать все палочки, и в случае положительного ответа предлагающий допустимую последовательность, в которой нужно собирать палочки.

33.5. *Разреженно-оболочечные распределения*

Рассмотрим задачу вычисления выпуклой оболочки множества заданных на плоскости точек, нанесенных в соответствии с каким-то известным случайным распределением. Для некоторых распределений математическое ожидание размера (количества точек) выпуклой оболочки такого множества, состоящего из n точек, равно $O(n^{1-\epsilon})$, где $\epsilon > 0$ — некоторая положительная константа. Назовем такое распределение ***разреженно-оболочечным*** (sparse-hulled). К разреженно-оболочечным распределениям относятся следующие.

- Точки, равномерно распределенные в круге единичного радиуса. Ожидаемый размер выпуклой оболочки равен $\Theta(n^{1/3})$.
- Точки, равномерно распределенные внутри выпуклого k-угольника, где k — произвольная константа. Ожидаемый размер выпуклой оболочки равен $\Theta(\lg n)$.
- Точки наносятся в соответствии с двумерным нормальным распределением. Ожидаемый размер выпуклой оболочки равен $\Theta(\sqrt{\lg n})$.

а. Даны два выпуклых многоугольника с n_1 и n_2 вершинами соответственно. Покажите, как построить выпуклую оболочку всех $n_1 + n_2$ точек за время $O(n_1 + n_2)$. (Многоугольники могут перекрываться.)

б. Покажите, как вычислить выпуклую оболочку n независимых точек, подчиненных некоторому разреженно-оболочечному распределению, за время $O(n)$. (*Указание:* рекурсивно постройте выпуклую оболочку для двух половин множества, а затем объедините полученные результаты.)

Заключительные замечания

В этой главе мы лишь вкратце обсудили тему алгоритмов и методов вычислительной геометрии. Среди серьезных изданий по вычислительной геометрии можно выделить книги Препараты (Preparata) и Шамоса (Shamos) [280], а также Эдельсбруннера (Edelsbrunner) [98] и О'Рурка (O'Rourke) [267].

Несмотря на то что геометрия изучается с античных времен, развитие алгоритмов для геометрических задач — относительно новое направление. Препарата и Шамос отмечают, что самое раннее упоминание о сложности задачи было сделано Э. Лемоном (E. Lemoine) в 1902 году. Изучая евклидовы построения,

в которых используются только циркуль и линейка, он свел все действия к набору пяти примитивов: размещение ножки циркуля в заданной точке, размещение ножки циркуля на заданной прямой, построение окружности, совмещение края линейки с заданной точкой и построение прямой. Лемона интересовало количество примитивов, необходимых для построения заданной конструкции; это число он назвал "простотой" данной конструкции.

Описанный в разделе 33.2 алгоритм, в котором определяется, пересекаются ли какие-либо отрезки, предложен Шамосом (Shamos) и Гоем (Hoey) [311].

Изначальная версия сканирования по Грэхему была представлена Грэхемом (Graham) [149]. Алгоритм оборачивания предложен Джарвисом (Jarvis) [188]. Яо (Yao) [357] с помощью модели дерева решений доказал, что нижняя граница времени работы алгоритма, предназначенного для построения произвольной выпуклой оболочки, равна $\Omega(n \lg n)$. С учетом количества вершин h выпуклой оболочки в асимптотическом пределе оптимальным является алгоритм отсечения и поиска, разработанный Киркпатриком (Kirkpatrick) и Зайделем (Seidel) [205], время работы которого равно $O(n \lg h)$.

Алгоритм "разделяй и властвуй" со временем работы $O(n \lg n)$, предназначенный для поиска пары ближайших точек, предложен Препаратой (Preparata) и опубликован в книге Препараты (Preparata) и Шамоса (Shamos) [280]. Они также показали, что в модели дерева решений этот алгоритм асимптотически оптимален.

Глава 34. NP-полнота

Почти все изученные нами ранее алгоритмы являются ***алгоритмами с полиномиальным временем работы*** (polynomial-time algorithms): для входных данных размером n их время работы в наихудшем случае равно $O(n^k)$, где k — некоторая константа. Возникает естественный вопрос: *все* ли задачи можно решить в течение полиномиального времени? Ответ отрицательный. В качестве примера можно привести знаменитую задачу останова, предложенную Тьюрингом (Turing). Ее невозможно решить ни на одном компьютере, каким бы количеством времени мы ни располагали. Существуют также задачи, которые можно решить, но не удается сделать это за время $O(n^k)$, где k — некоторая константа. Вообще говоря, о задачах, разрешимых с помощью алгоритмов с полиномиальным временем работы, возникает представление как о легко разрешимых или простых, а о задачах, время работы которых превосходит полиномиальное, — как о трудно разрешимых или сложных.

Однако темой этой главы является интересный класс задач, которые называются NP-полными, статус которых пока что неизвестен. Для решения NP-полных задач до настоящего времени не разработано алгоритмов с полиномиальным временем работы, но и не доказано, что для какой-либо из них такого алгоритма не существует. Этот так называемый вопрос $P \neq NP$ с момента своей постановки в 1971 году стал одним из самых трудных в теории вычислительных систем.

Особо интригующим аспектом NP-полных задач является то, что некоторые из них, на первый взгляд, аналогичны задачам, для решения которых существуют алгоритмы с полиномиальным временем работы. В каждой из описанных ниже пар задач одна из них разрешима в течение полиномиального времени, а другая является NP-полной. При этом различие между задачами кажется совершенно незначительным.

Поиск самых коротких и самых длинных простых путей. В главе 24 мы видели, что даже при отрицательных весах ребер *кратчайшие* пути от отдельно взятого источника в ориентированном графе $G = (V, E)$ можно найти за время $O(V\,E)$. Однако поиск *самого длинного* пути между двумя вершинами оказывается сложным. Задача определения того, содержит ли граф простой путь, количество ребер в котором не меньше заданного числа, является NP-полной.

Эйлеров и гамильтонов циклы. ***Эйлеров цикл*** (Euler tour) связанного ориентированного графа $G = (V, E)$ представляет собой цикл, в котором проход по

каждому *ребру* G осуществляется ровно один раз, хотя допускается неоднократное посещение некоторых вершин. В соответствии с результатами задачи 22.3 определить наличие эйлерова цикла (а также найти составляющие его ребра) можно за время $O(E)$. ***Гамильтонов цикл*** (hamiltonian cycle) ориентированного графа $G = (V, E)$ представляет собой простой цикл, содержащий все *вершины* из множества V. Задача определения, содержится ли в ориентированном графе гамильтонов цикл, является NP-полной. (Далее в этой главе будет доказано, что задача определения наличия гамильтонова цикла в *неориентированном* графе также является NP-полной.)

2-CNF- и 3-CNF-выполнимость. Булева формула содержит переменные, принимающие значения 0 и 1, булевы операторы, такие как $\wedge$ (И), $\vee$ (ИЛИ) и $\neg$ (НЕ), а также скобки. Булева формула называется ***выполнимой*** (satisfiable), если входящим в ее состав переменным можно присвоить такие значения 0 и 1, что в результате вычисления формулы получится значение 1. Далее в этой главе даются более формализованные определения всех терминов, а пока, говоря неформально, булева формула представлена в ***k-конъюнктивной нормальной форме*** (k-conjunctive normal form), или k-CNF, если она имеет вид конъюнкции (И) взятых в скобки выражений, являющихся дизъюнкцией (ИЛИ) ровно k переменных или их отрицаний (НЕ). Например, формула $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ представлена в 2-CNF. (Для нее существует выполнимый набор $x_1 = 1$, $x_2 = 0$, $x_3 = 1$.) Можно сформулировать алгоритм с полиномиальным временем работы, позволяющий определить, является ли 2-CNF-формула выполнимой. Однако, как будет показано далее в этой главе, определение, является ли 3-CNF-формула выполнимой, является NP-полной задачей.

NP-полнота и классы P и NP

В этой главе мы будем ссылаться на три класса задач: P, NP и NPC (класс NP-полных задач). В этом разделе они описываются неформально, а их формальное определение будет дано позже.

Класс P состоит из задач, разрешимых в течение полиномиального времени работы. Точнее говоря, это задачи, которые можно решить за время $O(n^k)$, где k — некоторая константа, а n — размер входных данных задачи. Большинство задач, рассмотренных в предыдущих главах, принадлежат классу P.

Класс NP состоит из задач, которые поддаются *проверке* в течение полиномиального времени. Имеется в виду, что если мы каким-то образом получаем "сертификат" решения, то в течение времени, полиномиальным образом зависящего от размера входных данных задачи, можно проверить корректность такого решения. Например, в задаче о гамильтоновом цикле с заданным ориентированным графом $G = (V, E)$ сертификат имел бы вид последовательности $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ из $|V|$ вершин. В течение полиномиального времени легко проверить, что $(v_i, v_{i+1}) \in E$ для $i = 1, 2, 3, \ldots, |V| - 1$ и что $(v_{|V|}, v_1) \in E$. Приведем другой пример: в задаче о 3-CNF-выполнимости в сертификате должно быть указано, какие значения следует присвоить переменным. В течение полиномиального времени легко проверить, удовлетворяет ли такое присваивание буле-

вой формуле. Любая задача класса P принадлежит классу NP, поскольку принадлежность задачи классу P означает, что ее решение можно получить в течение полиномиального времени, даже не располагая сертификатом. Это замечание будет формализовано ниже в данной главе, а пока что можно считать, что $P \subseteq NP$. Остается открытым вопрос, является ли P строгим подмножеством NP.

Неформально задача принадлежит классу NPC (такие задачи называются ***NP-полными*** (NP-complete)), если она принадлежит классу NP и является столь же "сложной", как и любая задача из класса NP. Ниже в этой главе будет дано формальное определение того, что означает выражение "задача такая же по сложности, как любая задача из класса NP". А пока что мы примем без доказательства положение, что если *любую* NP-полную задачу можно решить в течение полиномиального времени, то для *каждой* задачи из класса NP существует алгоритм с полиномиальным временем работы. Большинство ученых, занимающихся теорией вычислительных систем, считают NP-полные задачи очень трудноразрешимыми, потому что при огромном разнообразии изучавшихся до настоящего времени NP-полных задач ни для одной из них пока так и не найдено решение в виде алгоритма с полиномиальным временем работы. Таким образом, было бы крайне удивительно, если бы все они оказались разрешимыми в течение полиномиального времени.

Чтобы стать квалифицированным разработчиком алгоритмов, необходимо понимать основы теории NP-полноты. Если установлено, что задача NP-полная, это служит достаточно надежным указанием на то, что она трудноразрешимая. Как инженер вы эффективнее потратите время, если займетесь разработкой приближенного алгоритма (см. главу 35) или решением простого частного случая, вместо того чтобы искать быстрый алгоритм, выдающий точное решение задачи. Более того, многие естественно возникающие интересные задачи, которые, на первый взгляд, не сложнее задачи сортировки, поиска в графе или определения потока в сети, фактически являются NP-полными. Таким образом, важно ознакомиться с этим замечательным классом задач.

Как показать, что задача является NP-полной

Методы, позволяющие доказать, что та или иная задача является NP-полной, отличаются от методов, которые использовались в большей части этой книги для разработки и анализа алгоритмов. Имеется фундаментальная причина такого различия: чтобы показать, что задача является NP-полной, делается утверждение о том, насколько она сложна (или по крайней мере, насколько она сложна в нашем представлении), а не о том, насколько она проста. Не предпринимается попыток доказать, что существуют эффективные алгоритмы. Скорее, мы пытаемся доказать, что эффективных алгоритмов, вероятнее всего, не существует. В этом смысле доказательство NP-полноты несколько напоминает представленное в разделе 8.1 доказательство того, что нижняя граница любого алгоритма, работающего по методу сравнений, равна $\Omega(n \lg n)$. Однако методы, используемые для доказательства NP-полноты, отличаются от методов с применением дерева решений, описанных в разделе 8.1.

При доказательстве NP-полноты задачи используются три ключевые концепции, описанные ниже.

Задачи принятия решения и задачи оптимизации

Многие представляющие интерес задачи являются ***задачами оптимизации*** (optimization problems), каждому допустимому ("законному") решению которых можно сопоставить некоторое значение и для которых нужно найти допустимое решение с наилучшим значением. Например, в задаче поиска кратчайшего пути задаются неориентированный граф G, а также вершины u и v, и нужно найти путь из вершины u к вершине v, в котором содержится наименьшее количество ребер. (Другими словами, это задача поиска кратчайшего пути между парой вершин невзвешенного неориентированного графа.) Однако NP-полнота непосредственно применима не к задачам оптимизации, а к ***задачам принятия решения*** (decision problems), в которых ответ может быть положительным или отрицательным (говоря более формально, принимать значения "1" или "0").

Хотя при доказательстве NP-полноты задачи приходится ограничиваться задачами принятия решения, между ними и задачами оптимизации существует удобная взаимосвязь. Наложив ограничение на оптимизируемое значение, поставленную задачу оптимизации можно свести к соответствующей задаче принятия решения. Например, задаче поиска кратчайшего пути соответствует задача принятия решения о существовании пути: существует ли для заданных исходных данных, в число которых входит направленный граф G, вершины u и v и целое число k, путь из вершины u к вершине v, состоящий не более чем из k ребер?

Взаимосвязь между задачей оптимизации и соответствующей ей задачей принятия решения полезна для нас, если мы пытаемся показать, что задача оптимизации является "сложной". Причина этого заключается в том, что задача принятия решения в некотором смысле "проще", или по крайней мере "не сложнее". Например, задачу о существовании пути можно решить, решив задачу поиска кратчайшего пути, а затем сравнив количество ребер в найденном кратчайшем пути со значением параметра k в задаче принятия решения. Другими словами, если задача оптимизации простая, соответствующая ей задача принятия решения также простая. Формулируя это утверждение так, чтобы оно имело большее отношение к NP-полноте, можно доказать, что если удается засвидетельствовать сложность задачи принятия решения, это означает, что соответствующая задача оптимизации также сложная. Таким образом, хотя и приходится ограничиваться рассмотрением задач принятия решения, теория NP-полноты зачастую имеет следствия и для задач оптимизации.

Приведение

Сделанное выше замечание о том, что одна задача не сложнее или не легче другой, применимо, даже если обе задачи являются задачами принятия решения. Эта идея используется почти во всех доказательствах NP-полноты. Это делается следующим образом. Рассмотрим задачу принятия решения, скажем, задачу A, которую хотелось бы решить в течение полиномиального времени. Назовем

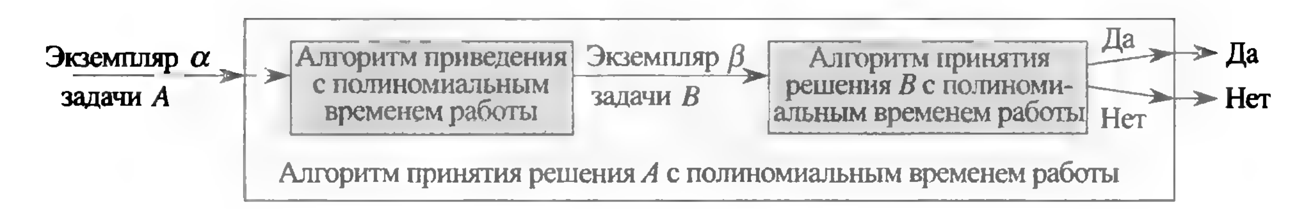


Рис. 34.1. Использование алгоритма приведения с полиномиальным временем работы для решения задачи принятия решения A за полиномиальное время при наличии алгоритма с полиномиальным временем работы, решающего некоторую другую задачу B. Экземпляр α задачи A преобразуется за полиномиальное время в экземпляр β задачи B, после чего задача B решается за полиномиальное время, и ответ для экземпляра β используется как ответ для экземпляра α.

входные данные отдельно взятой задачи ***экземпляром*** (instance) этой задачи. Например, экземпляром задачи существования пути является некоторый граф G, некоторые его вершины u и v, а также некоторое целое число k. А теперь предположим, что существует другая задача принятия решения, скажем, B, для которой заранее известно, как решить ее в течение полиномиального времени. Наконец предположим, что имеется процедура с приведенными ниже характеристиками, преобразующая любой экземпляр α задачи A в некоторый экземпляр β задачи B.

- Преобразование выполняется за полиномиальное время.
- Ответы являются идентичными, т.е. в экземпляре α ответ "да" выдается тогда и только тогда, когда в экземпляре β также выдается ответ "да".

Назовем такую процедуру ***алгоритмом приведения*** (reduction algorithm) с полиномиальным временем. Как видно из рис. 34.1, эта процедура предоставляет описанный ниже способ решения задачи A за полиномиальное время.

1. Заданный экземпляр α задачи A с помощью алгоритма приведения с полиномиальным временем преобразуется в экземпляр β задачи B.
2. Запускается алгоритм, решающий экземпляр β задачи принятия решения B в течение полиномиального времени.
3. Ответ для экземпляра β используется в качестве ответа для экземпляра α.

Поскольку для выполнения каждого из перечисленных выше этапов требуется полиномиальное время, это относится и ко всему процессу в целом, что дает способ решения экземпляра α задачи за полиномиальное время. Другими словами, путем приведения задачи A к задаче B "простота" задачи B используется для доказательства "простоты" задачи A.

Если вспомнить, что при доказательстве NP-полноты требуется показать, насколько сложной, а не насколько простой является задача, доказательство NP-полноты с помощью приведения с полиномиальным временем работы выполняется обратно описанному выше способу. Продвинемся в разработке этой идеи еще на шаг и посмотрим, как с помощью приведения с полиномиальным временем показать, что для конкретной задачи B не существует алгоритмов с полиномиальным временем работы. Предположим, что имеется задача принятия решения A, относительно которой заранее известно, что для нее не существует алгоритма с по-

линомиальным временем работы. (Пока что мы не станем обременять себя размышлениями о том, как сформулировать такую задачу A.) Предположим также, что имеется преобразование, позволяющее в течение полиномиального времени преобразовать экземпляры задачи A в экземпляры задачи B. Теперь с помощью простого доказательства "от противного" можно показать, что для решения задачи B не существует алгоритмов с полиномиальным временем работы. Предположим обратное, т.е. что существует решение задачи B в виде алгоритма с полиномиальным временем работы. Тогда, воспользовавшись методом, проиллюстрированным на рис. 34.1, можно получить способ решения задачи A в течение полиномиального времени, а это противоречит предположению о том, что таких алгоритмов для задачи A не существует.

Если речь идет о NP-полноте, то здесь нельзя предположить, что для задачи A вообще не существует алгоритмов с полиномиальным временем работы. Однако методология аналогична в том отношении, что доказательство NP-полноты задачи B основывается на предположении об NP-полноте задачи A.

Первая NP-полная задача

Поскольку метод приведения основан на том, что для какой-то задачи заранее известна ее NP-полнота, то для доказательства NP-полноты различных задач понадобится "первая" NP-полная задача. В качестве таковой воспользуемся задачей, в которой задана булева комбинационная схема, состоящая из логических элементов И, ИЛИ и НЕ. В задаче спрашивается, существует ли для этой схемы такой набор входных булевых величин, для которого будет выдано значение 1. Доказательство NP-полноты этой первой задачи будет представлено в разделе 34.3.

Краткое содержание главы

В этой главе изучаются аспекты NP-полноты, которые непосредственно основываются на анализе алгоритмов. В разделе 34.1 формализуется понятие "задачи" и определяется класс сложности P как класс задач принятия решения, разрешимых в течение полиномиального времени. Также станет ясно, как эти понятия укладываются в рамки теории формальных языков. В разделе 34.2 определяется класс NP, к которому относятся задачи принятия решения, правильность решения которых можно проверить в течение полиномиального времени. Здесь также формально ставится вопрос $\mathrm{P} \neq \mathrm{NP}$.

В разделе 34.3 показано, как с помощью приведения с полиномиальным временем работы изучаются взаимоотношения между задачами. Здесь дано определение NP-полноты и представлен набросок доказательства того, что задача "о выполнимости схемы" является NP-полной. Имея одну NP-полную задачу, в разделе 34.4 мы увидим, как можно существенно проще доказать NP-полноту других задач с помощью приведения. Эта методология проиллюстрирована на примере двух задач на выполнимость формул, для которых доказывается NP-полнота. В разделе 34.5 NP-полнота демонстрируется для широкого круга других задач.

34.1. Полиномиальное время

Начнем изучение NP-полноты с формализации понятия задач, разрешимых в течение полиномиального времени. Эти задачи считаются легко разрешимыми, но не из математических, а из философских соображений. В поддержку этого мнения можно привести три аргумента.

Во-первых, хотя и разумно считать трудноразрешимой задачу, для решения которой требуется время $\Theta(n^{100})$, на практике крайне редко встречаются задачи, время решения которых выражается полиномом такой высокой степени. Для практических задач, которые решаются за полиномиальное время, показатель степени обычно намного меньше. Опыт показывает, что если для задачи становится известен алгоритм с полиномиальным временем работы, то зачастую впоследствии разрабатывается и более эффективный алгоритм. Даже если самый лучший из известных на сегодняшний день алгоритмов решения задачи характеризуется временем $\Theta(n^{100})$, достаточно высока вероятность того, что вскоре будет разработан алгоритм с намного лучшим временем работы.

Во-вторых, для многих приемлемых вычислительных моделей задача, которая решается в течение полиномиального времени в одной модели, может быть решена в течение полиномиального времени и в другой. Например, в большей части книги рассматривается класс задач, разрешимых в течение полиномиального времени с помощью последовательных машин с произвольным доступом к памяти. Этот класс совпадает с классом задач, разрешимых в течение полиномиального времени на абстрактных машинах Тьюринга[1]. Он также совпадает с классом задач, разрешимых в течение полиномиального времени на параллельных компьютерах, если зависимость количества процессоров от объема входных данных описывается полиномиальной функцией.

В-третьих, класс задач, разрешимых в течение полиномиального времени, обладает полезными свойствами замкнутости, поскольку множество полиномов замкнуто относительно операций сложения, умножения и композиции. Например, если выход одного алгоритма с полиномиальным временем работы соединить со входом другой такой задачи, получится полиномиальный составной алгоритм. В упр. 34.1.5 требуется показать, что если алгоритм с полиномиальным временем работы фиксированное количество раз вызывает подпрограммы с полиномиальным временем работы и выполняет дополнительные действия также за полиномиальное время, то и общее время работы такого составного алгоритма является полиномиальным.

Абстрактные задачи

Чтобы получить представление о классе задач, разрешимых за полиномиальное время, сначала необходимо формально описать само понятие "задача".

[1]Подробное рассмотрение модели Тьюринга можно найти в книгах Хопкрофта (Hopcroft) и Ульмана (Ullman) [179], а также Льюиса (Lewis) и Пападимитриу (Papadimitriou) [235].

Определим ***абстрактную задачу*** (abstract problem) Q как бинарное отношение между множеством ***экземпляров*** (instances) задач I и множеством ***решений*** (solutions) задач S. Например, экземпляр задачи о поиске кратчайшего пути SHORTEST-PATH состоит из трех элементов: графа и двух вершин. Решением этой задачи является последовательность вершин графа; при этом пустая последовательность может означать, что искомого пути не существует. Сама задача SHORTEST-PATH представляет собой отношение, сопоставляющее каждому экземпляру графа и двум его вершинам кратчайший путь по графу, соединяющий эти две вершины. Поскольку кратчайший путь может быть не единственным, конкретный экземпляр задачи может иметь несколько решений.

Представленная выше формулировка абстрактной задачи носит более широкий характер, чем требуется для наших целей. Как мы уже видели, теория NP-полноты ограничивается рассмотрением ***задач принятия решения*** (decision problems), решения которых имеют вид "да/нет". В этом случае абстрактную задачу принятия решения можно рассматривать как функцию, отображающую экземпляр множества I на множество решений $\{0, 1\}$. Например, задача принятия решения, соответствующая задаче SHORTEST-PATH, — рассмотренная выше задача поиска пути PATH. Если $i = \langle G, u, v, k\rangle$ — экземпляр задачи принятия решения PATH, то равенство $\text{PATH}(i) = 1$ (да) выполняется, если количество ребер в кратчайшем пути из вершины u в вершину v не превышает k; в противном случае $\text{PATH}(i) = 0$ (нет). Многие абстрактные задачи являются не задачами принятия решения, а ***задачами оптимизации*** (optimization problems), в которых некоторое значение подлежит минимизации или максимизации. Однако ранее мы убедились, что обычно не составляет труда сформулировать задачу оптимизации как задачу принятия решения, которая не сложнее исходной.

Кодирование

Если абстрактная задача решается с помощью компьютерной программы, экземпляры задач необходимо представить в виде, понятном этой программе. ***Кодирование*** (encoding) множества S абстрактных объектов — это отображение e множества S на множество бинарных строк[2]. Например, всем известно, что натуральные числа $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ кодируются строками $\{0, 1, 10, 11, 100, \ldots\}$. В этой кодировке $e(17) = 10001$. Каждый, кто интересовался, как в компьютере представляются символы клавиатуры, вероятно, знаком с кодами ASCII. В кодировке ASCII представление символа `A` выглядит как 1000001. В виде бинарной строки можно закодировать даже составной объект. Для этого конструируется комбинация, состоящая из представлений элементов, которые содержит в себе этот объект. Многоугольники, графы, функции, ориентированные пары, программы — все это можно закодировать бинарными строками.

Таким образом, компьютерный алгоритм, который "решает" некоторую абстрактную задачу принятия решения, фактически принимает в качестве вход-

---

[2]Область значений отображения e — это необязательно *бинарные* строки; подойдет любое множество строк, состоящих из символов конечного алфавита, содержащего не менее двух символов.

ных данных закодированный экземпляр задачи. Назовем задачу, множество экземпляров которой является множеством бинарных строк, ***конкретной*** (concrete problem). Говорят, что алгоритм ***решает*** (solves) конкретную задачу в течение времени $O(T(n))$, если для заданного экземпляра задачи i длиной $n = |i|$ с его помощью можно получить решение за время $O(T(n))$[3]. Поэтому конкретная задача ***разрешима в течение полиномиального времени*** (polynomial-time solvable), если существует алгоритм, позволяющий решить ее за время $O(n^k)$ для некоторой константы k.

Теперь можно формально определить ***класс сложности*** **P** (complexity class P) как множество конкретных задач принятия решения, разрешимых за полиномиальное время.

С помощью кодирования абстрактные задачи можно отображать на конкретные. Данную абстрактную задачу принятия решения Q, отображающую множество экземпляров I на множество $\{0,1\}$, с помощью кодирования $e : I \to \{0,1\}^*$ можно свести к связанной с ней конкретной задаче принятия решения, которая будет обозначаться как $e(Q)$[4]. Если $Q(i) \in \{0,1\}$ — решение экземпляра абстрактной задачи $i \in I$, то оно же является и решением экземпляра конкретной задачи $e(i) \in \{0,1\}^*$. Заметим, что некоторые бинарные строки могут не представлять осмысленного экземпляра абстрактной задачи. Для удобства мы будем предполагать, что любая такая строка произвольным образом отображается на 0. Таким образом, конкретная задача имеет те же решения, что и абстрактная, если ее экземпляры в виде бинарных строк представляют закодированные экземпляры абстрактной задачи.

Было бы неплохо расширить определение разрешимости в течение полиномиального времени для конкретных задач на абстрактные задачи, воспользовавшись в качестве связующего звена кодами; однако хотелось бы, чтобы определение не зависело от вида кодировки. Другими словами, эффективность решения задачи не должна зависеть от того, как она кодируется. К сожалению, на самом деле существует достаточно сильная зависимость от кодирования. Например, предположим, что в качестве входных данных алгоритма выступает только целое число k и что время работы этого алгоритма равно $\Theta(k)$. Если число k передается как ***унарное*** (unary), т.е. в виде строки, состоящей из k единиц, то время работы алгоритма для входных данных длины n равно $O(n)$ (выражается полиномиальной функцией). Если же используется более естественное бинарное представление числа k, то длина входной строки равна $n = \lfloor \lg k \rfloor + 1$. В этом случае время работы алгоритма равно $\Theta(k) = \Theta(2^n)$, т.е. зависит от объема входных данных как показательная функция. Таким образом, в зависимости от способа кодирования алгоритм может как выполняться за полиномиальное время, так и иметь время работы, превосходящее полиномиальное.

---

[3]Предполагается, что выходные данные алгоритма отделены от его входных данных. Поскольку для получения каждого выходного бита требуется по крайней мере один элементарный временной интервал, а всего имеется $O(T(n))$ временных интервалов, объем выходных данных представляет собой $O(T(n))$.

[4]Мы обозначаем через $\{0,1\}^*$ множество всех строк, составленных из символов множества $\{0,1\}$.

Поэтому кодирование абстрактной задачи — достаточно важный вопрос для нашего понимания полиномиального времени. Невозможно вести речь о решении абстрактной задачи, не представив подробного описания кодировки. Тем не менее, если отбросить такие "дорогостоящие" коды, как унарные, само кодирование задачи на практике будет мало влиять на то, разрешима ли задача в течение полиномиального времени. Например, если представить целые числа в троичной системе счисления, а не в двоичной, это не повлияет на то, разрешима ли задача в течение полиномиального времени, поскольку целое число в троичной системе счисления можно преобразовать в целое число в двоичной системе счисления за полиномиальное время.

Говорят, что функция $f : \{0,1\}^* \to \{0,1\}^*$ ***вычислима за полиномиальное время*** (polynomial-time computable), если существует алгоритм A с полиномиальным временем работы, который для произвольных входных данных $x \in \{0,1\}^*$ возвращает выходные данные $f(x)$. Для некоторого множества I экземпляров задач две кодировки, e_1 и e_2, называются ***полиномиально связанными*** (polynomially related), если существуют такие две вычислимые в течение полиномиального времени функции, f_{12} и f_{21}, что для любого экземпляра $i \in I$ выполняются равенства $f_{12}(e_1(i)) = e_2(i)$ и $f_{21}(e_2(i)) = e_1(i)$[5]. Другими словами, закодированную величину $e_2(i)$ можно вычислить на основе закодированной величины $e_1(i)$ с помощью алгоритма с полиномиальным временем работы и наоборот. Если две кодировки, e_1 и e_2, абстрактной задачи полиномиально связаны, то, как следует из представленной ниже леммы, разрешимость задачи в течение полиномиального времени не зависит от используемой кодировки.

Лемма 34.1

Пусть Q — абстрактная задача принятия решения, определенная на множестве экземпляров I, а e_1 и e_2 — полиномиально связанные кодировки множества I. В этом случае $e_1(Q) \in \mathrm{P}$ тогда и только тогда, когда $e_2(Q) \in \mathrm{P}$.

Доказательство. Достаточно доказать только прямое утверждение, поскольку обратное утверждение симметрично по отношению к прямому. Поэтому предположим, что задачу $e_1(Q)$ можно решить за время $O(n^k)$, где k — некоторая константа. Далее, предположим, что для любого экземпляра i задачи кодирование $e_1(i)$ можно получить из кодирования $e_2(i)$ за время $O(n^c)$, где c — некоторая константа, а $n = |e_2(i)|$. Чтобы решить задачу $e_2(Q)$ с входными данными $e_2(i)$, сначала вычисляется $e_1(i)$, после чего алгоритм запускается для решения задачи $e_1(Q)$ с входными данными $e_1(i)$. Сколько времени это займет? Для преобразования кодировки требуется время $O(n^c)$, поэтому выполняется равенство $|e_1(i)| = O(n^c)$, поскольку объем выходных данных алгоритма на последователь-

---

[5]Кроме того, накладывается техническое требование, чтобы функции f_{12} и f_{21} "отображали неэкземпляры в неэкземпляры". ***Неэкземпляр*** (noninstance) в кодировке e — это строка $x \in \{0,1\}^*$, такая, что не существует экземпляра i, для которого $e(i) = x$. Потребуем, чтобы равенство $f_{12}(x) = y$ выполнялось для каждого неэкземпляра x в кодировке e_1, где y — некоторый неэкземпляр в кодировке e_2, а равенство $f_{21}(x') = y'$ — для каждого неэкземпляра x' в кодировке e_2, где y' — некоторый неэкземпляр в кодировке e_1.

ном компьютере не может превосходить по величине время его работы. Решение задачи с входными данными $e_1(i)$ занимает время $O(|e_1(i)|^k) = O(n^{ck})$, которое является полиномиальным, поскольку и c, и k — константы. ∎

Таким образом, то, как закодированы экземпляры абстрактной задачи — в двоичной системе счисления или в троичной, — не влияет на ее "сложность", т.е. на то, разрешима ли она в течение полиномиального времени. Однако, если эти экземпляры имеют унарную кодировку, сложность задачи может измениться. Чтобы иметь возможность осуществлять преобразование независимым от кодировки образом, в общем случае предполагается, что экземпляры задачи закодированы в произвольном рациональном и сжатом виде, если специально не оговаривается противное. Точнее говоря, предполагается, что кодирование целых чисел полиномиально связано с их бинарным представлением и что кодирование ограниченного множества полиномиально связано с его кодированием в виде списка элементов, заключенных в скобки и разделенных запятыми. (Одна из таких схем кодирования — ASCII.) Располагая таким "стандартным" кодом, можно получить рациональный код других математических объектов, таких как кортежи, графы и формулы. Для обозначения стандартного кода объекта этот объект будет заключаться в угловые скобки. Таким образом, $\langle G \rangle$ обозначает стандартный код графа G.

До тех пор, пока неявно используется код, полиномиально связанный с таким стандартным кодом, можно прямо говорить об абстрактных задачах, не ссылаясь при этом на какой-то отдельный код, зная, что выбор кода не повлияет на разрешимость абстрактной задачи в течение полиномиального времени. С этого момента в общем случае предполагается, что все экземпляры задачи представляют собой бинарные строки, закодированные с помощью стандартного кодирования, если явно не оговаривается противное. Кроме того, в большинстве случаев мы будем пренебрегать различием между абстрактными и конкретными задачами. Однако на практике читателю следует остерегаться тех возникающих на практике задач, в которых стандартное кодирование не очевидно и выбор кодирования имеет значение.

Структура формальных языков

Один из аргументов в пользу удобства задач принятия решения заключается в том, что для них можно использовать алгоритмы из теории формальных языков. Здесь сто́ит привести некоторые определения из этой теории. ***Алфавит* Σ** (alphabet Σ) представляет собой конечное множество символов. ***Языком* L** (language L), определенным над множеством Σ, является произвольное множество строк, состоящих из символов из множества Σ. Например, если $\Sigma = \{0, 1\}$, то множество $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ является языком бинарного представления простых чисел. Обозначим ***пустую строку*** (empty string) как ε, а ***пустой язык*** (empty language) — как $\emptyset$. Язык всех строк, заданных над множеством Σ, обозначается как Σ^*. Например, если $\Sigma = \{0, 1\}$, то $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ — множество всех бинарных строк. Любой язык L над множеством Σ является подмножеством множества Σ^*.

Над языками можно определить ряд операций. Наличие таких операций из теории множеств, как ***объединение*** (union) и ***пересечение*** (intersection), непосредственно следует из теоретико-множественной природы определения языков. ***Дополнение*** (complement) языка L определим с помощью соотношения $\overline{L} = \Sigma^* - L$. ***Конкатенацией*** (concatenation) языков L_1 и L_2 является язык

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ и } x_2 \in L_2\} \ .$$

Замыканием (closure), или ***замыканием Клини*** (Kleene star), языка L называется язык

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots \ ,$$

где L^k — язык, полученный в результате k-кратной конкатенации языка L с самим собой.

С точки зрения теории языков множество экземпляров любой задачи принятия решения Q — это просто множество Σ^*, где $\Sigma = \{0, 1\}$. Поскольку множество Q полностью характеризуется теми экземплярами задачи, в которых выдается ответ 1 (да), это множество можно рассматривать как язык L над множеством $\Sigma = \{0, 1\}$, где

$$L = \{x \in \Sigma^* : Q(x) = 1\} \ .$$

Например, задаче принятия решения PATH соответствует язык

$$\begin{aligned} \text{PATH} = \{\langle G, u, v, k\rangle : \ & G = (V, E) \text{ — неориентированный граф,} \\ & u, v \in V, \\ & k \geq 0 \text{ — целое число, и} \\ & \text{в } G \text{ имеется путь из } u \text{ в } v\text{, состоящий} \\ & \text{не более чем из } k \text{ ребер}\} \ . \end{aligned}$$

(Для удобства иногда одно и то же имя (в данном случае это имя PATH) будет употребляться и для задачи принятия решения, и для соответствующего ей языка.)

Схема формальных языков позволяет в сжатом виде выразить взаимоотношение между задачами принятия решения и решающими их алгоритмами. Говорят, что алгоритм A ***принимает*** (accepts) строку $x \in \{0, 1\}^*$, если для заданных входных данных x выход алгоритма $A(x)$ равен 1. Язык, ***принимаемый*** (accepted) алгоритмом A, представляет собой множество строк $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, т.е. множество строк, принимаемых алгоритмом. Алгоритм A ***отвергает*** (rejects) строку x, если $A(x) = 0$.

Даже если язык L принимается алгоритмом A, этот алгоритм необязательно отвергает строку $x \notin L$, предоставляемую в качестве его входных данных. Например, алгоритм может быть организован в виде бесконечного цикла. Язык L ***распознается*** (decided) алгоритмом A, если каждая бинарная строка этого языка принимается алгоритмом A, а каждая бинарная строка, которая не принадлежит языку L, отвергается этим алгоритмом. Язык L ***принимается за полиномиальное время*** (accepted in polynomial time) алгоритмом A, если он принимается алгоритмом A и, кроме того, если существует такая константа k, что для любой n-символьной строки $x \in L$ алгоритм A принимает строку x за время $O(n^k)$. Язык

L ***распознается за полиномиальное время*** (decided in polynomial time) алгоритмом A, если существует такая константа k, что для любой n-символьной строки $x \in \{0,1\}^*$ алгоритм за время $O(n^k)$ правильно выясняет, принадлежит ли строка x языку L. Таким образом, чтобы принять язык, алгоритму нужно заботиться только о строках языка L, а чтобы распознать язык, он должен корректно принять или отвергнуть каждую строку из множества $\{0,1\}^*$.

Как пример, — язык PATH может быть принят в течение полиномиального времени. Один из принимающих алгоритмов с полиномиальным временем работы проверяет, закодирован ли объект G как неориентированный граф, убеждается, что u и v — его вершины, с помощью поиска в ширину находит в графе G кратчайший путь из вершины u к вершине v, а затем сравнивает количество ребер в полученном кратчайшем пути с числом k. Если в объекте G закодирован неориентированный граф и путь из вершины u к вершине v содержит не более k ребер, алгоритм выводит значение 1 и останавливается. В противном случае он работает бесконечно долго. Однако такой алгоритм не распознает язык PATH, поскольку он не выводит явно значение 0 для экземпляров, длина кратчайшего пути в которых превышает k ребер. Предназначенный для задачи PATH алгоритм распознавания должен явно отвергать бинарные строки, не принадлежащие языку PATH. Для такой задачи принятия решения, как задача PATH, подобный алгоритм распознавания разработать легко: вместо того чтобы работать бесконечно долго, если не существует пути из вершины u в вершину v, количество ребер в котором не превышает k, этот алгоритм должен выводить значение 0 и останавливаться. Для других задач, таких как задача останова Тьюринга, принимающий алгоритм существует, а алгоритма распознавания не существует.

Можно неформально определить ***класс сложности*** (complexity class) как множество языков, принадлежность к которому определяется ***мерой сложности*** (complexity measure), такой, как время работы алгоритма, определяющего, принадлежит ли данная строка x языку L. Фактическое определение класса сложности носит более технический характер[6].

Воспользовавшись описанным выше формализмом теории языков, можно дать альтернативное определение класса сложности P:

$$\text{P} = \{L \subseteq \{0,1\}^* : \text{существует алгоритм } A \text{, разрешающий язык } L \text{ за полиномиальное время}\} .$$

Фактически P является также классом языков, которые могут быть приняты за полиномиальное время.

Теорема 34.2

$$\text{P} = \{L : L \text{ принимается алгоритмом с полиномиальным временем работы}\} .$$

[6]Интересующийся читатель найдет дополнительную информацию в статье Хартманиса (Hartmanis) и Стирнса (Stearns) [161].

Доказательство. Поскольку класс языков, которые распознаются алгоритмами с полиномиальным временем работы, представляет собой подмножество класса языков, которые принимаются алгоритмами с полиномиальным временем работы, остается лишь показать, что если язык L принимается алгоритмом с полиномиальным временем работы, то он также распознается алгоритмом с полиномиальным временем работы. Пусть L — язык, который принимается некоторым алгоритмом с полиномиальным временем работы A. Воспользуемся классическим "модельным" доказательством, чтобы сконструировать другой алгоритм с полиномиальным временем работы A', который бы распознавал язык L. Поскольку алгоритм A принимает язык L за время $O(n^k)$, где k — некоторая константа, существует такая константа c, что алгоритм A принимает язык L не более чем за cn^k шагов. Для любой входной строки x алгоритм A' моделирует cn^k шагов алгоритма A. После этого алгоритм A' проверяет поведение алгоритма A. Если он принял строку x, то алгоритм A' также принимает эту строку, выводя значение 1. Если же алгоритм A не принял строку x, то алгоритм A' отвергает эту строку, выводя значение 0. Накладные расходы алгоритма A', моделирующего алгоритм A, приводят к увеличению времени работы не более чем на полиномиальный множитель, поэтому A' — алгоритм с полиномиальным временем работы, распознающий язык L. ■

Заметим, что доказательство теоремы 34.2 является неконструктивным. Для данного языка $L \in \text{P}$ граница времени работы алгоритма A, который принимает язык L, на самом деле может быть неизвестна. Тем не менее известно, что такая граница существует, поэтому существует алгоритм A', способный проверить эту границу, даже если его не всегда удается легко найти.

Упражнения

34.1.1

Определим задачу оптимизации LONGEST-PATH-LENGTH как отношение, связывающее каждый экземпляр задачи, состоящий из неориентированного графа и двух его вершин, с количеством ребер в самом длинном простом пути между этими двумя вершинами. Определим задачу принятия решения LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ — неориентированный граф, $u, v \in V$, $k \geq 0$ — целое число, и существует простой путь из u в v в G, состоящий как минимум из k ребер$\}$. Покажите, что задачу оптимизации LONGEST-PATH-LENGTH можно решить за полиномиальное время тогда и только тогда, когда LONGEST-PATH-LENGTH $\in$ P.

34.1.2

Дайте формальное определение задачи поиска самого длинного простого цикла в неориентированном графе. Сформулируйте соответствующую задачу принятия решения. Опишите язык, соответствующий этой задаче принятия решения.

34.1.3

Разработайте формальное кодирование ориентированного графа в виде бинарных строк для представления с помощью матриц смежности. Выполните то же самое для представления с использованием списка смежных вершин. Докажите, что эти два представления полиномиально связаны.

34.1.4

Является ли алгоритм динамического программирования для решения дискретной задачи о рюкзаке, который предлагалось разработать в упр. 16.2.2, алгоритмом с полиномиальным временем работы? Обоснуйте свой ответ.

34.1.5

Покажите, что алгоритм, который содержит не больше некоторого постоянного количества вызовов процедуры с полиномиальным временем работы, сам работает полиномиальное время. Если же алгоритм делает полиномиальное число вызовов такой процедуры, то общее время работы может быть экспоненциальным.

34.1.6

Покажите, что класс P, который рассматривается как множество языков, замкнут относительно операций объединения, пересечения, конкатенации, дополнения и замыкания Клини. Другими словами, если $L_1, L_2 \in \text{P}$, то $L_1 \cup L_2 \in \text{P}$, $L_1 \cap L_2 \in \text{P}$, $L_1 L_2 \in \text{P}$, $\overline{L}_1 \in \text{P}$ и $L_1^* \in \text{P}$.

34.2. Проверка за полиномиальное время

Теперь рассмотрим алгоритм, проверяющий принадлежность языку. Например, предположим, что в данном экземпляре $\langle G, u, v, k \rangle$ задачи принятия решения PATH задан также путь p из вершины u в вершину v. Легко проверить, превышает ли длина пути p величину k. Если она не превышает эту величину, путь p можно рассматривать как "сертификат" того, что данный экземпляр действительно принадлежит PATH. Для задачи принятия решения PATH такой сертификат, по-видимому, не дает ощутимых преимуществ. В конце концов, эта задача принадлежит классу P (фактически ее можно решить за линейное время), поэтому проверка принадлежности с помощью такого сертификата занимает столько же времени, сколько и решение задачи. А теперь исследуем задачу, для которой пока неизвестен алгоритм принятия решения с полиномиальным временем работы, но если имеется сертификат, то выполнить его проверку очень легко.

Гамильтоновы циклы

Задача поиска гамильтоновых циклов в неориентированном графе изучается уже более ста лет. Формально ***гамильтонов цикл*** (hamiltonian cycle) неориентированного графа $G = (V, E)$ представляет собой простой цикл, содержащий все вершины множества V. Граф, содержащий гамильтонов цикл, называют ***га-***

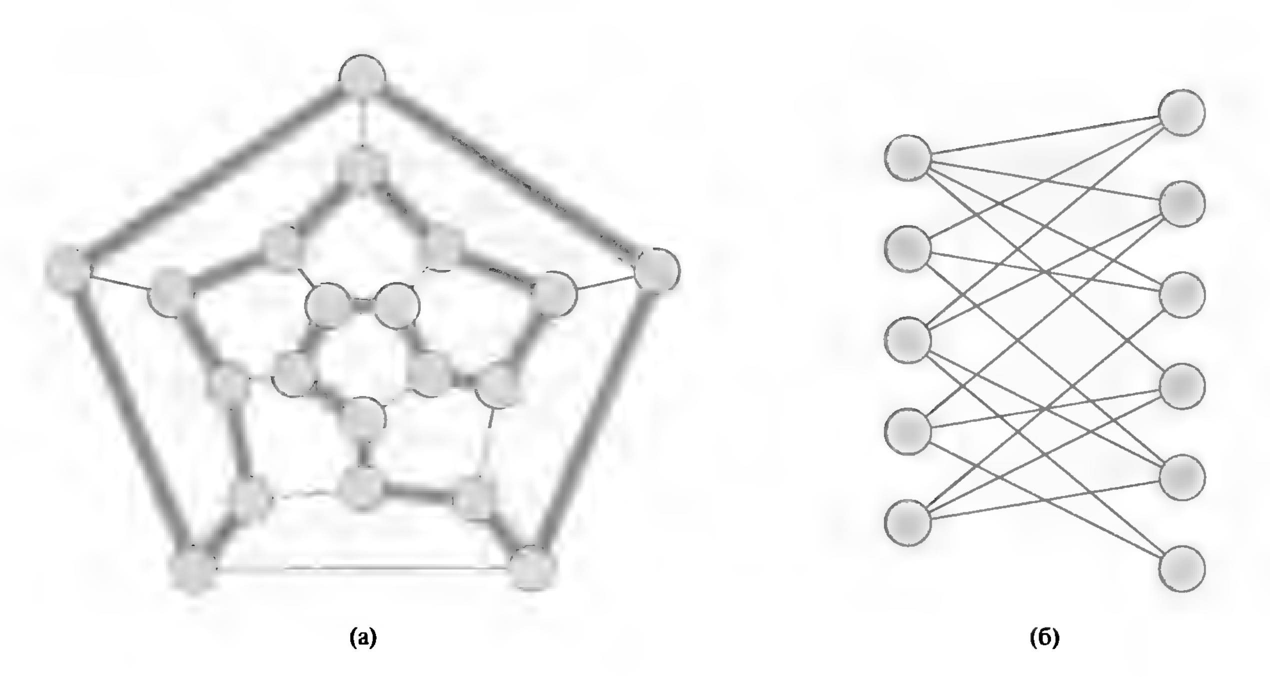


Рис. 34.2. **(а)** Граф, представляющий вершины, ребра и грани додекаэдра; штриховкой показан один из его гамильтоновых циклов. **(б)** Двудольный граф с нечетным количеством вершин. Все такие графы негамильтоновы.

мильтоновым (hamiltonian); в противном случае он является ***негамильтоновым*** (nonhamiltonian). Он назван так в честь У.Р. Гамильтона (W.R. Hamilton), который описал математическую игру на додекаэдре (рис. 34.2, (а)), в которой один игрок отмечает пятью булавками пять произвольных последовательных вершин, а другой игрок должен так дополнить этот путь, чтобы в результате получился путь, содержащий все вершины[7]. Додекаэдр является гамильтоновым графом, и один из гамильтоновых циклов показан на рис. 34.2, (а). Однако не все графы являются гамильтоновыми. Например, на рис. 34.2, (б) показан двудольный граф с нечетным количеством вершин. В упр. 34.2.2 предлагается доказать, что все такие графы негамильтоновы.

Задачу о гамильтоновых циклах (hamiltonian-cycle problem), в которой нужно выяснить, содержит ли граф G гамильтонов цикл, можно определить как формальный язык:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ является гамильтоновым графом}\} \ .$$

Как алгоритм мог бы распознать язык HAM-CYCLE? Для заданного экземпляра задачи $\langle G \rangle$ можно предложить алгоритм принятия решения, который сначала формировал бы список всех перестановок вершин графа G, а затем проверял каждую перестановку, — не является ли она гамильтоновым путем? Чему рав-

[7]В письме от 17 октября 1856 года своему другу Джону Грейвзу (John T. Graves) Гамильтон пишет [156, p. 624]: "Я обнаружил, что некоторых молодых людей весьма позабавила новая математическая игра, в которой один человек ставит пять кнопок в любые пять последовательных точек ...а другой игрок затем пытается вставить остальные пятнадцать кнопок (что согласно теории, изложенной в данном письме, всегда возможно) так, чтобы получился цикл, так, чтобы были охвачены все точки, и чтобы он закончился рядом с точкой, с которой начал противник".

нялось бы время работы такого алгоритма? При использовании "рационального" кодирования графа с использованием матриц смежности количество вершин m графа равно $\Omega(\sqrt{n})$, где $n = |\langle G \rangle|$ — длина кода графа G. Всего имеется $m!$ возможных перестановок вершин, поэтому время работы алгоритма равно $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, и оно не ведет себя в асимптотическом пределе как $O(n^k)$ ни для какой константы k. Таким образом, время работы подобного прямолинейного алгоритма не выражается полиномиальной функцией. Фактически, как будет доказано в разделе 34.5, задача о гамильтоновых циклах является NP-полной.

Алгоритмы верификации

Рассмотрим несколько упрощенную задачу. Предположим, что приятель сообщил вам, что данный граф G — гамильтонов, а затем предложил доказать это, предоставив последовательность вершин, образующих гамильтонов цикл. Очевидно, в такой ситуации доказательство было бы достаточно простым: следует просто проверить, является ли предоставленный цикл гамильтоновым, убедившись, что данная последовательность вершин является перестановкой множества всех вершин V и что каждое встречающееся в цикле ребро действительно принадлежит графу. Легко понять, что подобный алгоритм верификации можно реализовать так, чтобы время его работы было равно $O(n^2)$, где n — длина графа G в закодированном виде. Таким образом, доказательство того, что гамильтонов цикл в графе существует, можно проверить за полиномиальное время.

Определим ***алгоритм верификации*** (verification algorithm) как алгоритм A с двумя аргументами, один из которых является обычной входной строкой x, а второй — бинарной строкой y под названием ***сертификат*** (certificate). Двухаргументный алгоритм A ***верифицирует*** (verifies) входную строку x, если существует сертификат y, такой, что $A(x, y) = 1$. ***Язык, верифицированный*** алгоритмом верификации A, представляет собой множество

$$L = \{x \in \{0,1\}^* : \text{ существует } y \in \{0,1\}^*, \text{ такое, что } A(x,y) = 1\} \ .$$

Интуитивно понятно, что алгоритм A верифицирует язык L, если для любой строки $x \in L$ существует сертификат y, позволяющий доказать с помощью алгоритма A, что $x \in L$. Кроме того, для любой строки $x \notin L$ не должно существовать сертификата, доказывающего, что $x \in L$. Например, в задаче о гамильтоновом цикле сертификатом является список вершин некоторого гамильтонового цикла. Если граф гамильтонов, то гамильтонов цикл сам по себе предоставляет достаточно информации для верификации этого факта. В обратном случае, т.е. если граф не является гамильтоновым, не существует списка вершин, позволяющего обмануть алгоритм верификации и установить, что граф является гамильтоновым, так как алгоритм верификации выполняет тщательную проверку предложенного "цикла", чтобы убедиться в его правильности.

Класс сложности NP

Класс сложности **NP** (complexity class NP) — это класс языков, которые можно верифицировать с помощью алгоритма с полиномиальным временем работы[8]. Точнее говоря, язык L принадлежит классу NP тогда и только тогда, когда существуют алгоритм A с двумя входными параметрами и полиномиальным временем работы, а также константа c, такая, что

$$L = \{x \in \{0,1\}^* : \text{существует сертификат } y \text{ с } |y| = O(|x|^c), \text{ такой, что } A(x,y) = 1\} .$$

При этом говорят, что алгоритм A ***верифицирует*** язык L ***за полиномиальное время***.

Из проведенного ранее обсуждения задачи о гамильтоновых циклах следует, что задача HAM-CYCLE $\in$ NP (всегда приятно знать, что некоторое важное множество — не пустое). Кроме того, если $L \in$ P, то $L \in$ NP, поскольку при наличии алгоритма с полиномиальным временем работы, способного распознать язык L, алгоритм легко преобразовать в алгоритм верификации с двумя аргументами, который просто игнорирует сертификат и принимает именно те входные строки, для которых он устанавливает принадлежность языку L. Таким образом, P $\subseteq$ NP.

Неизвестно, выполняется ли равенство P $=$ NP, но, по мнению большинства исследователей, классы P и NP — не одно и то же. Интуитивно понятно, что класс P состоит из задач, которые решаются быстро. Класс же NP состоит из задач, решение которых можно быстро проверить. Возможно, из своего опыта вы уже знаете, что часто сложнее решить задачу с самого начала, чем проверить представленное решение, особенно если вы ограничены во времени. Среди ученых, занимающихся теорией вычислительных систем, распространено мнение, что эта аналогия распространяется и на классы P и NP, а следовательно, что класс NP содержит языки, не принадлежащие классу P.

Существует более веское, хотя и не окончательное, свидетельство того, что P $\neq$ NP, — наличие языков, являющихся "NP-полными". Класс этих задач рассматривается в разделе 34.3.

Помимо вопроса о выполнении соотношения P $\neq$ NP, остаются нерешенными и многие другие фундаментальные вопросы. На рис. 34.3 показаны некоторые возможные сценарии. Несмотря на интенсивные исследования в этой области, никому не удалось установить, замкнут ли класс NP относительно операции дополнения. Другими словами, следует ли из соотношения $L \in$ NP соотношение $\overline{L} \in$ NP? Можно определить ***класс сложности*** **co-NP** (complexity class co-NP) как множество языков L, таких, что $\overline{L} \in$ NP. Вопрос о том, замкнут ли класс NP относительно дополнения, можно перефразировать как вопрос о выполнении равенства NP $=$ co-NP. Поскольку класс P замкнут относительно допол-

---

[8]Название "NP" означает "nondeterministic polynomial time" (недетерминистическое полиномиальное время). Класс NP изначально изучался в контексте недетерминизма, но нами используется более простое, хотя и эквивалентное понятие верификации. В книге Хопкрофта (Hopcroft) и Ульмана (Ullman) [179] хорошо изложено понятие NP-полноты в терминах недетерминистических вычислительных моделей.

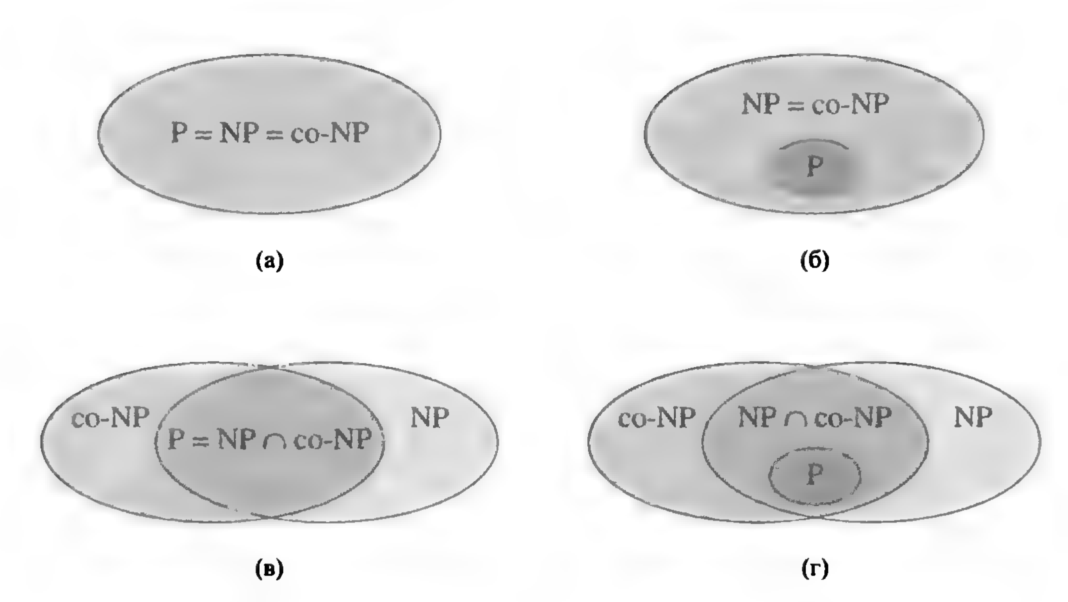


Рис. 34.3. Четыре возможных вида отношений между классами сложности. На представленной диаграмме полное включение одной области в другую указывает на отношение истинного подмножества. **(а)** $P = NP = \text{co-NP}$. Большинство исследователей полагают, что эта возможность наименее правдоподобна. **(б)** Если NP замкнут относительно дополнения, то $NP = \text{co-NP}$, но при этом не обязательно выполняется соотношение $P = NP$. **(в)** $P = NP \cap \text{co-NP}$, но NP не замкнут относительно дополнения. **(г)** $NP \neq \text{co-NP}$ и $P \neq NP \cap \text{co-NP}$. Большинство исследователей полагают эту возможность наиболее правдоподобной.

нения (упр. 34.1.6), из упр. 34.2.9 следует, что $P \subseteq NP \cap \text{co-NP}$. Однако, опять же, неизвестно, выполняется ли равенство $P = NP \cap \text{co-NP}$, или в множестве $NP \cap \text{co-NP} - P$ существует ли хотя бы один язык.

Таким образом, к сожалению, наше понимание того, какие именно взаимоотношения существуют между классами P и NP, далеко не полное. Тем не менее, несмотря на то что мы неспособны доказать сложность конкретной задачи, доказав ее NP-полноту, можно получить о ней очень важную информацию.

Упражнения

34.2.1

Рассмотрим язык GRAPH-ISOMORPHISM $= \{\langle G_1, G_2 \rangle : G_1$ и G_2 являются изоморфными графами$\}$. Докажите, что GRAPH-ISOMORPHISM $\in$ NP, описав алгоритм с полиномиальным временем работы, позволяющий верифицировать этот язык.

34.2.2

Докажите, что если G — неориентированный двудольный граф с нечетным количеством вершин, то он не является гамильтоновым.

34.2.3

Покажите, что если HAM-CYCLE $\in$ P, то задача о выводе списка вершин гамильтонового цикла в порядке их обхода разрешима в течение полиномиального времени.

34.2.4

Докажите, что класс языков NP замкнут относительно операций объединения, пересечения, конкатенации и замыкания Клини. Обсудите замкнутость класса NP относительно дополнения.

34.2.5

Покажите, что любой язык из класса NP можно распознать с помощью алгоритма, время работы которого равно $2^{O(n^k)}$, где k — некоторая константа.

34.2.6

Гамильтонов путь (hamiltonian path) графа представляет собой простой путь, который проходит через каждую вершину ровно по одному разу. Покажите, что язык HAM-PATH $= \{\langle G, u, v\rangle$: в графе G имеется гамильтонов путь из u в $v\}$ принадлежит NP.

34.2.7

Покажите, что задачу о гамильтоновом пути в ориентированном ациклическом графе из упр. 34.2.6 можно решить в течение полиномиального времени. Сформулируйте эффективный алгоритм ее решения.

34.2.8

Пусть ϕ — булева формула, составленная из булевых входных переменных x_1, $x_2, \ldots, x_k$, операторов НЕ ($\neg$), И ($\wedge$), ИЛИ ($\vee$) и скобок. Формула ϕ называется ***тавтологией*** (tautology), если для всех возможных наборов входных переменных в результате вычисления формулы получается значение 1. Определите язык булевых формул TAUTOLOGY, состоящий из тавтологий. Покажите, что TAUTOLOGY $\in$ co-NP.

34.2.9

Докажите, что P $\subseteq$ co-NP.

34.2.10

Докажите, что если NP $\neq$ co-NP, то P $\neq$ NP.

34.2.11

Пусть G — связный неориентированный граф, содержащий не менее трех вершин, а G^3 — граф, полученный путем соединения всех пар вершин, которые связаны в графе G путем, длина которого не превышает трех ребер. Докажите, что граф G^3 гамильтонов. (*Указание:* постройте остовное дерево графа G и воспользуйтесь индукцией.)

34.3. NP-полнота и приводимость

Пожалуй, одна из наиболее веских причин, по которым специалисты в области теории вычислительных систем полагают, что P $\neq$ NP, — наличие класса "NP-

полных" задач. Этот класс обладает замечательным свойством, которое состоит в том, что если хоть одну (*любую*) NP-полную задачу можно решить в течение полиномиального времени, то и *все* задачи этого класса обладают решением за полиномиальное время, т.е. P = NP. Однако, несмотря на многолетние исследования, до сих пор не обнаружено ни одного алгоритма с полиномиальным временем работы ни для одной NP-полной задачи.

Язык HAM-CYCLE — одна из NP-полных задач. Если бы этот язык можно было распознать за полиномиальное время, то каждую задачу из класса NP можно было бы решить в течение полиномиального времени. Фактически, если бы множество NP − P оказалось непустым, то с уверенностью можно было бы утверждать, что HAM-CYCLE ∈ NP − P.

В определенном смысле NP-полные языки — "самые сложные" в классе NP. В этом разделе будет показано, как относительная "сложность" языков сравнивается с помощью точного понятия под названием "приводимость за полиномиальное время". Затем приводится формальное определение NP-полных языков, а в конце раздела дается набросок доказательства того, что один из таких языков — CIRCUIT-SAT — является NP-полным. В разделах 34.4 и 34.5 с помощью понятия приводимости демонстрируется, что многие другие задачи также являются NP-полными.

Приводимость

Интуитивно понятно, что задачу Q можно свести к другой задаче Q', если любой экземпляр задачи Q "легко перефразируется" в экземпляр задачи Q', решение которого позволяет получить решение соответствующего экземпляра задачи Q. Например, задача решения линейных уравнений относительно неизвестной величины x сводится к задаче решения квадратных уравнений. Если задан экземпляр $ax + b = 0$, его можно преобразовать в уравнение $0x^2 + ax + b = 0$, решение которого совпадает с решением уравнения $ax + b = 0$. Таким образом, если задача Q сводится к другой задаче Q', то решить задачу Q в некотором смысле "не сложнее", чем задачу Q'.

Возвращаясь к применению системы формальных языков для решения задач, будем говорить, что язык L_1 ***приводим за полиномиальное время*** (polynomial-time reducible) к языку L_2 (что обозначается как $L_1 \leq_P L_2$, если существует вычислимая за полиномиальное время функция $f : \{0,1\}^* \to \{0,1\}^*$, такая что для всех $x \in \{0,1\}^*$

$$x \in L_1 \text{ тогда и только тогда, когда } f(x) \in L_2 \ . \qquad (34.1)$$

Функцию f называют ***функцией приведения*** (reduction function), а алгоритм F с полиномиальным временем работы, вычисляющий функцию f, — ***алгоритмом приведения*** (reduction algorithm).

На рис. 34.4 проиллюстрировано приведение языка L_1 к другому языку L_2 за полиномиальное время. Каждый язык — это подмножество множества $\{0,1\}^*$. Функция приведения f обеспечивает такое отображение за полиномиальное время, что если $x \in L_1$, то $f(x) \in L_2$. Кроме того, если $x \notin L_1$, то $f(x) \notin L_2$. Таким

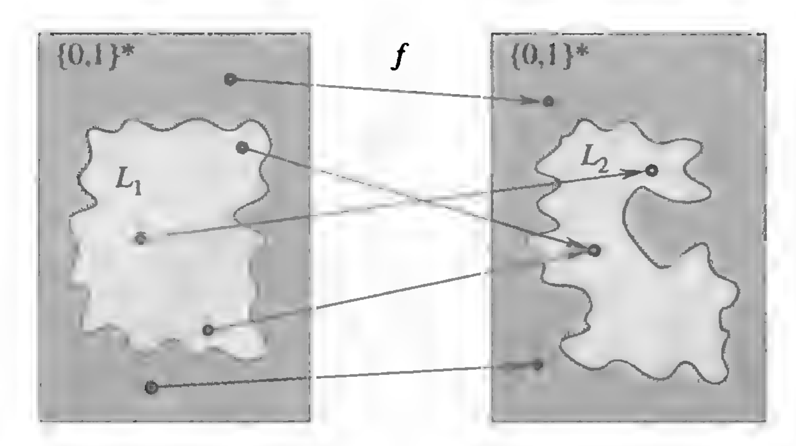


Рис. 34.4. Иллюстрация к приведению за полиномиальное время языка L_1 к языку L_2 с помощью функции приведения f. Для любых входных данных $x \in \{0,1\}^*$ вопрос о том, справедливо ли соотношение $x \in L_1$, имеет тот же ответ, что и вопрос о справедливости $f(x) \in L_2$.

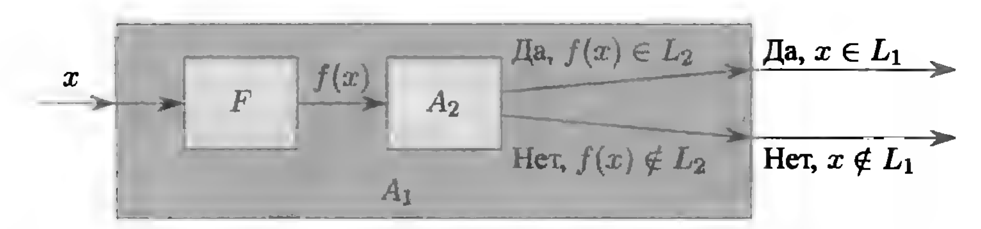


Рис. 34.5. Доказательство леммы 34.3. Алгоритм F представляет собой алгоритм приведения, который вычисляет функцию приведения f из L_1 в L_2 за полиномиальное время, а A_2 — алгоритм с полиномиальным временем работы, разрешающий язык L_2. Алгоритм A_1 выясняет справедливость $x \in L_1$, используя F для преобразования любого входа x в $f(x)$, а затем применяет A_2, чтобы выяснить справедливость $f(x) \in L_2$.

образом, функция приведения отображает любой экземпляр x задачи принятия решения, представленной языком L_1, на экземпляр $f(x)$ задачи, представленной языком L_2. Ответ на вопрос о том, принадлежит ли экземпляр $f(x)$ языку L_2, непосредственно позволяет ответить на вопрос о том, принадлежит ли экземпляр x языку L_1.

Приведение за полиномиальное время служит мощным инструментом доказательства того, что различные языки принадлежат классу P.

Лемма 34.3

Если $L_1, L_2 \subseteq \{0,1\}^*$ являются языками, такими, что $L_1 \leq_{\text{P}} L_2$, то из $L_2 \in \text{P}$ следует $L_1 \in \text{P}$.

Доказательство. Пусть A_2 — алгоритм с полиномиальным временем работы, который распознает язык L_2, а F — алгоритм приведения с полиномиальным временем работы, вычисляющий функцию приведения f. Построим алгоритм A_1 с полиномиальным временем работы, который распознает язык L_1.

На рис. 34.5 проиллюстрирован процесс построения алгоритма A_1. Для заданного набора входных данных $x \in \{0,1\}^*$ в алгоритме A_1 с помощью алгоритма F входные данные x преобразуются в $f(x)$, после чего с помощью алгоритма A_2 проверяется, выполняется ли $f(x) \in L_2$. Результат работы алгоритма A_2 выводится алгоритмом A_1 в качестве ответа.

Корректность алгоритма A_1 следует из условия (34.1). Алгоритм выполняется за полиномиальное время, поскольку и алгоритм F, и алгоритм A_2 завершают свою работу за полиномиальное время (см. упр. 34.1.5). ■

NP-полнота

Приведения за полиномиальное время служат формальным средством, позволяющим показать, что одна задача по сложности такая же, как и другая, по крайней мере с точностью до полиномиально-временного множителя. То есть, если $L_1 \leq_P L_2$, сложность языка L_1 превышает сложность языка L_2 не более чем на полиномиальный множитель. Вот почему отношение "меньше или равно" для приведения является мнемоническим. Теперь можно определить множество NP-полных языков, являющихся самыми сложными задачами класса NP.

Язык $L \subseteq \{0,1\}^*$ является ***NP-полным*** (NP-complete), если

1. $L \in \text{NP}$;
2. $L' \leq_P L$ для каждого $L' \in \text{NP}$.

Если язык L удовлетворяет свойству 2, но не обязательно удовлетворяет свойству 1, говорят, что L — ***NP-сложный*** (NP-hard). Определим также класс NPC как класс NP-полных языков.

Как показано в приведенной далее теореме, NP-полнота — ключевая проблема в разрешении вопроса о том, равны ли на самом деле классы P и NP.

Теорема 34.4

Если некоторая NP-полная задача разрешима за полиномиальное время, то $\text{P} = \text{NP}$. Это эквивалентно утверждению о том, что если какая-нибудь задача из класса NP не решается за полиномиальное время, то ни одна из NP-полных задач не решается за полиномиальное время.

Доказательство. Предположим, что $L \in \text{P}$ и что $L \in \text{NPC}$. Для любого языка $L' \in \text{NP}$ согласно свойству 2 определения NP-полноты справедливо $L' \leq_P L$. Таким образом, в соответствии с леммой 34.3 мы также имеем $L' \in \text{P}$, что и доказывает первое утверждение теоремы.

Чтобы доказать второе утверждение, заметим, что оно является противоположностью первого. ■

Вот почему при исследовании вопроса $\text{P} \neq \text{NP}$ внимание акцентируется на NP-полных задачах. Большинство специалистов по теории вычислительных систем полагают, что $\text{P} \neq \text{NP}$, так что отношения между классами P, NP и NPC являются такими, как показано на рис. 34.6. Но если кто-нибудь додумается до алгоритма с полиномиальным временем работы, способного решить NP-полную задачу, тем самым будет доказано, что $\text{P} = \text{NP}$. Однако поскольку до сих пор такой алгоритм не обнаружен ни для одной NP-полной задачи, доказательство NP-полноты задачи является веским свидетельством того, что ее трудно решить.

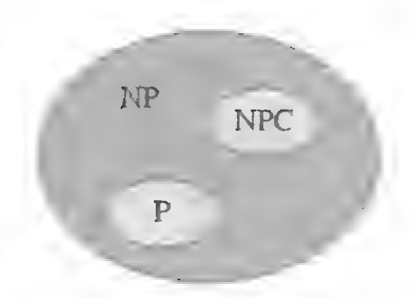


Рис. 34.6. Представление большинства специалистов в области информатики об отношении между классами P, NP и NPC: классы P и NPC полностью содержатся в классе NP, и $P \cap NPC = \emptyset$.

Выполнимость схем

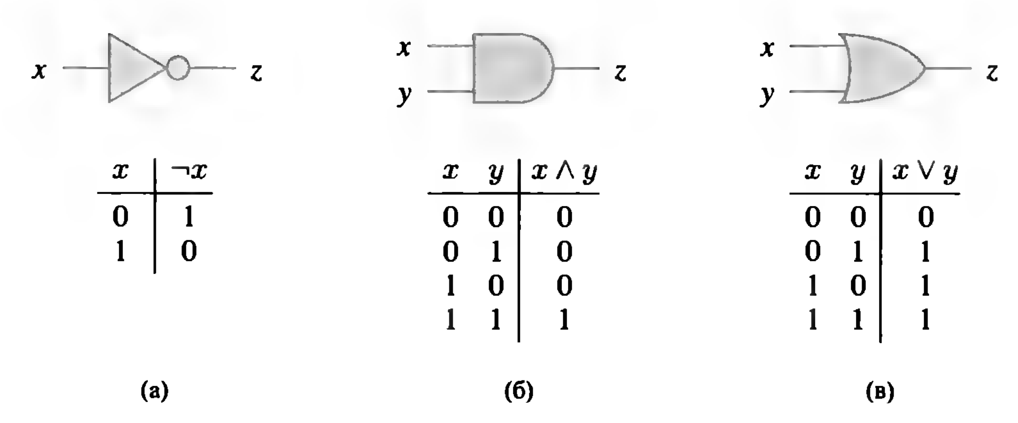
Хотя определение NP-полноты нами уже сформулировано, до сих пор не была доказана NP-полнота ни одной задачи. Как только это будет сделано хотя бы для одной задачи, с помощью приводимости в течение полиномиального времени можно будет доказать NP-полноту других задач. Таким образом, сосредоточим внимание на том, чтобы продемонстрировать NP-полноту задачи о выполнимости схем.

К сожалению, для формального доказательства того, что задача о выполнимости схем является NP-полной, требуются технические детали, выходящие за рамки настоящей книги. Вместо этого дадим неформальное описание доказательства, основанного на базовом понимании булевых комбинационных схем.

Булевы комбинационные схемы составляются из булевых комбинационных элементов, соединенных между собой проводами. ***Булев комбинационный элемент*** (boolean combinational element), или ***комбинационный логический элемент***, — это любой элемент сети, обладающий фиксированным количеством булевых входов и выходов, который выполняет вполне определенную функцию. Булевы значения выбираются из множества $\{0, 1\}$, где 0 представляет значение FALSE, а 1 — значение TRUE.

Комбинационные логические элементы, используемые в задаче о выполнимости схем, вычисляют простую булеву функцию и известны как ***логические элементы*** (logic gates). На рис. 34.7 показаны три основных логических элемента, использующихся в задаче о выполнимости схем: ***логический элемент НЕ*** (NOT gate), или ***инвертор*** (inverter), ***логический элемент И*** (AND gate) и ***логический элемент ИЛИ*** (OR gate). На элемент НЕ поступает одна бинарная ***входная величина*** (input) x, значение которой равно 0 или 1; элемент выдает бинарную ***выходную величину*** (output) z, значение которой противоположно значению входной величины. На каждый из двух других элементов поступают две бинарные входные величины, x и y, а в результате получается одна бинарная выходная величина, z.

Работу каждого логического и каждого комбинационного элемента можно описать ***таблицей истинности*** (truth table), представленной под каждым элементом на рис. 34.7. В таблице истинности для всех возможных наборов входных величин перечисляются выходные значения данного комбинационного элемента. Например, в таблице истинности элемента ИЛИ показано, что если его входные значения $x = 0$ и $y = 1$, то выходное значение равно $z = 1$. Для обозначения



x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Рис. 34.7. Три основных логических элемента с бинарными входами и выходами. Для каждого элемента приведена таблица истинности, описывающая его работу. **(а)** Элемент НЕ. **(б)** Элемент И. **(в)** Элемент ИЛИ.

функции НЕ используется символ $\neg$, для обозначения функции И — символ $\wedge$, а для обозначения функции ИЛИ — символ $\vee$. Например, $0 \vee 1 = 1$.

Элементы И и ИЛИ можно обобщить на случай, когда входных значений больше двух. Выходное значение элемента И равно 1, если все его входные значения равны 1; в противном случае его выходное значение равно 0. Выходное значение элемента ИЛИ равно 1, если хотя бы одно из его входных значений равно 1; в противном случае его выходное значение равно 0.

Булева комбинационная схема (boolean combinational circuit) состоит из одного или нескольких комбинационных логических элементов, соединенных ***проводами*** (wires). Провод может соединять выход одного элемента со входом другого, подавая таким образом выходное значение первого элемента на вход второго. На рис. 34.8 изображены две похожие одна на другую булевы комбинационные схемы, отличающиеся лишь одним элементом. В части (а) рисунка также приводятся значения, которые передаются по каждому проводу для входа $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$. Хотя к одному и тому же проводу нельзя подключить более одного вывода комбинационного элемента, сигнал с него может поступать одновременно на входы нескольких элементов. Количество элементов, для которых данный провод предоставляет входные данные, называется его ***коэффициентом ветвления*** (fan-out). Если к проводу не подсоединены выходы никаких элементов, он называется ***входом схемы*** (circuit input), получающим входные данные из внешних источников. Если к проводу не подсоединен вход ни одного элемента, он называется ***выходом схемы*** (circuit output), по которому выдаются результаты работы схемы. (Ответвление внутреннего провода также может выступать в роли выхода.) Чтобы определить задачу о выполнимости схемы, следует ограничиться рассмотрением схем с одним выходом, хотя на практике при разработке аппаратного оборудования встречаются логические комбинационные схемы с несколькими выводами.

Логическая комбинационная схема не содержит циклов. Поясним это нагляднее. Предположим, что схеме сопоставляется ориентированный граф $G = (V, E)$ с вершинами в каждом комбинационном элементе и k ориентированными ребрами для каждого провода, коэффициент разветвления которых равен k. Ориентированное ребро (u, v) в этом графе существует, если провод соединяет выход

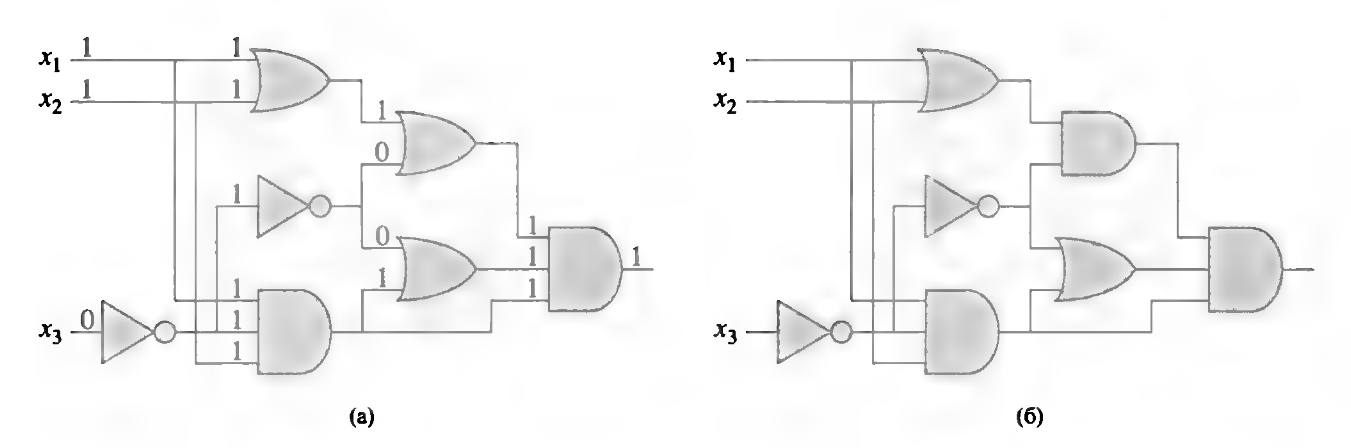


Рис. 34.8. Два экземпляра задачи о выполнимости схем. **(а)** Входные значения $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ приводят к появлению на выходе 1. Таким образом, эта схема выполнима. **(б)** Никакие входные значения не приводят к появлению 1 на выходе данной схемы. Таким образом, эта схема невыполнима.

элемента u со входом элемента v. Полученный в результате такого построения граф должен быть ациклическим.

Будем рассматривать ***наборы значений*** (truth assignment) булевых переменных, соответствующих входам схемы. Говорят, что логическая комбинационная схема ***выполнима*** (satisfiable), если она имеет ***выполняющий набор*** (satisfying assignment): набор значений, в результате подачи которого на вход схемы ее выходное значение равно 1. Например, для схемы, изображенной на рис. 34.8,(а), выполняющий набор имеет вид $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$; следовательно, эта схема выполнима. В упр. 34.3.1 предлагается показать, что никакое присваивание значений величинам x_1, x_2 и x_3 не приведет к значению 1 на выходе схемы, изображенной на рис. 34.8,(б). Эта схема всегда выдает значение 0, поэтому она невыполнима.

Задача о выполнимости схемы (circuit-satisfiability problem) формулируется следующим образом: выполнима ли заданная логическая комбинационная схема, состоящая из элементов И, ИЛИ и НЕ. Однако, чтобы поставить этот вопрос формально, необходимо принять соглашение по поводу стандартного кода для таких схем. ***Размер*** (size) логической комбинационной схемы определяется как сумма количества логических комбинационных элементов и числа проводов в схеме. Можно предложить код, используемый при представлении графов, который отображает каждую заданную схему C на бинарную строку $\langle C\rangle$, длина которой выражается не более чем полиномиальной функцией от размера схемы. Таким образом, можно определить формальный язык

$$\text{CIRCUIT-SAT} = \{\langle C\rangle : C \text{ является выполнимой булевой схемой}\} \ .$$

Задача о выполнимости схем возникает при оптимизации компьютерного аппаратного обеспечения. Если какая-то подсхема рассматриваемой схемы всегда выдает значение 0, то ее можно заменить более простой подсхемой, в которой опускаются все логические элементы, а на выход подается постоянное значение 0. Было бы полезно иметь алгоритм решения этой задачи, время которого выражалось бы полиномиальной функцией.

36 Зак. 3726

Чтобы проверить, разрешима ли данная схема C, можно попытаться просто перебрать все возможные комбинации входных значений. К сожалению, если в схеме k входов, то количество таких комбинаций равно 2^k. Если размер схемы C выражается полиномиальной функцией от k, то проверка всех комбинаций занимает время $\Omega(2^k)$, а эта функция по скорости роста превосходит полиномиальную функцию[9]. Как уже упоминалось, имеется веский аргумент в пользу того, что не существует алгоритмов с полиномиальным временем работы, способных решить задачу о выполнимости схем, так как эта задача является NP-полной. Разобьем доказательство NP-полноты этой задачи на две части, соответствующие двум частям определения NP-полноты.

Лемма 34.5

Задача о выполнимости схем принадлежит классу NP.

Доказательство. В этом доказательстве будет сформулирован алгоритм A с двумя входными параметрами и полиномиальным временем работы, способный проверять решение задачи CIRCUIT-SAT. В роли одного из входных параметров алгоритма A выступает логическая комбинационная схема C (точнее, ее стандартный код). Вторым входным параметром является сертификат, который представляет собой булевы значения в проводах схемы. (См. другой сертификат, меньшего объема, в упр. 34.3.4.)

Алгоритм A строится следующим образом. Для каждого логического элемента схемы проверяется, что предоставляемое сертификатом значение на выходном проводе правильно вычисляется как функция значений на входных проводах. Далее, если на выход всей цепи подается значение 1, алгоритм также выдает значение 1, поскольку величины, поступившие на вход схемы C, являются выполняющим набором. В противном случае алгоритм выводит значение 0.

Для любой выполнимой схемы C, выступающей в роли входного параметра алгоритма A, существует сертификат, длина которого выражается полиномиальной функцией от размера схемы C, и который приводит к тому, что на выход алгоритма A подается значение 1. Для любой невыполнимой схемы, выступающей в роли входного параметра алгоритма A, никакой сертификат не может заставить этот алгоритм поверить в то, что эта схема выполнима. Алгоритм A выполняется за полиномиальное время: при хорошей реализации достаточно будет линейного времени. Таким образом, задачу CIRCUIT-SAT можно проверить за полиномиальное время, и CIRCUIT-SAT $\in$ NP. ∎

Вторая часть доказательства NP-полноты задачи CIRCUIT-SAT заключается в том, чтобы показать, что ее язык является NP-сложным. Другими словами, необходимо показать, что каждый язык класса NP приводится за полиномиальное

---

[9] С другой стороны, если размер схемы C равен $\Theta(2^k)$, то алгоритм со временем работы $O(2^k)$ завершается по истечении времени, выражаемого полиномиальной функцией от размера схемы. Даже если $P \neq NP$, эта ситуация не противоречит NP-полноте задачи; из наличия алгоритма с полиномиальным временем работы для частного случая не следует, что такой алгоритм существует в общем случае.

время к языку CIRCUIT-SAT. Само доказательство этого факта содержит много технических сложностей, поэтому здесь приводится набросок доказательства, основанный на некоторых принципах работы аппаратного обеспечения компьютера.

Компьютерная программа хранится в памяти компьютера в виде последовательности инструкций. Типичная инструкция содержит код операции, которую нужно выполнить, адреса операндов в памяти и адрес, куда следует поместить результат. Специальная ячейка памяти под названием ***счетчик команд*** (program counter — PC) следит за тем, какая инструкция должна выполняться следующей. При извлечении очередной инструкции показание счетчика команд автоматически увеличивается на единицу. Это приводит к последовательному выполнению инструкций компьютером. Однако в результате выполнения определенных инструкций в счетчик команд может быть записано некоторое значение, что приводит к изменению обычного последовательного порядка выполнения. Таким образом организуются циклы и условные ветвления.

В любой момент выполнения программы состояние вычислений в целом можно представить содержимым памяти компьютера. (Имеется в виду область памяти, состоящая из самой программы, счетчика команд, рабочей области и всех других битов состояния, с помощью которых компьютер ведет учет выполнения программы.) Назовем любое отдельное состояние памяти компьютера ***конфигурацией*** (configuration). Выполнение команд можно рассматривать как отображение одной конфигурации на другую. Важно то, что аппаратное обеспечение, осуществляющее это отображение, можно реализовать в виде логической комбинационной схемы, которая при доказательстве приведенной ниже леммы будет обозначена как M.

Лемма 34.6

Задача о выполнимости схем является NP-сложной.

Доказательство. Пусть L — язык класса NP. Опишем алгоритм F с полиномиальным временем работы, вычисляющий функцию приведения f, которая отображает каждую бинарную строку x на схему $C = f(x)$, такую, что $x \in L$ тогда и только тогда, когда $C \in$ CIRCUIT-SAT.

Поскольку $L \in$ NP, должен существовать алгоритм A, верифицирующий язык L за полиномиальное время. В алгоритме F, который будет построен ниже, для вычисления функции приведения f будет использован алгоритм A с двумя входными параметрами.

Пусть $T(n)$ — время работы алгоритма A в наихудшем случае для входной строки длиной n, $k \geq 1$ — константа, такая, что $T(n) = O(n^k)$, а длина сертификата равна $O(n^k)$. (В действительности время работы алгоритма A выражается полиномиальной функцией от полного объема входных данных, в состав которых входят и входная строка, и сертификат, но так как длина сертификата полиномиальным образом зависит от длины n входной строки, время работы алгоритма полиномиально зависит от n.)

Основная идея доказательства заключается в том, чтобы представить выполнение алгоритма A в виде последовательности конфигураций. Как видно из

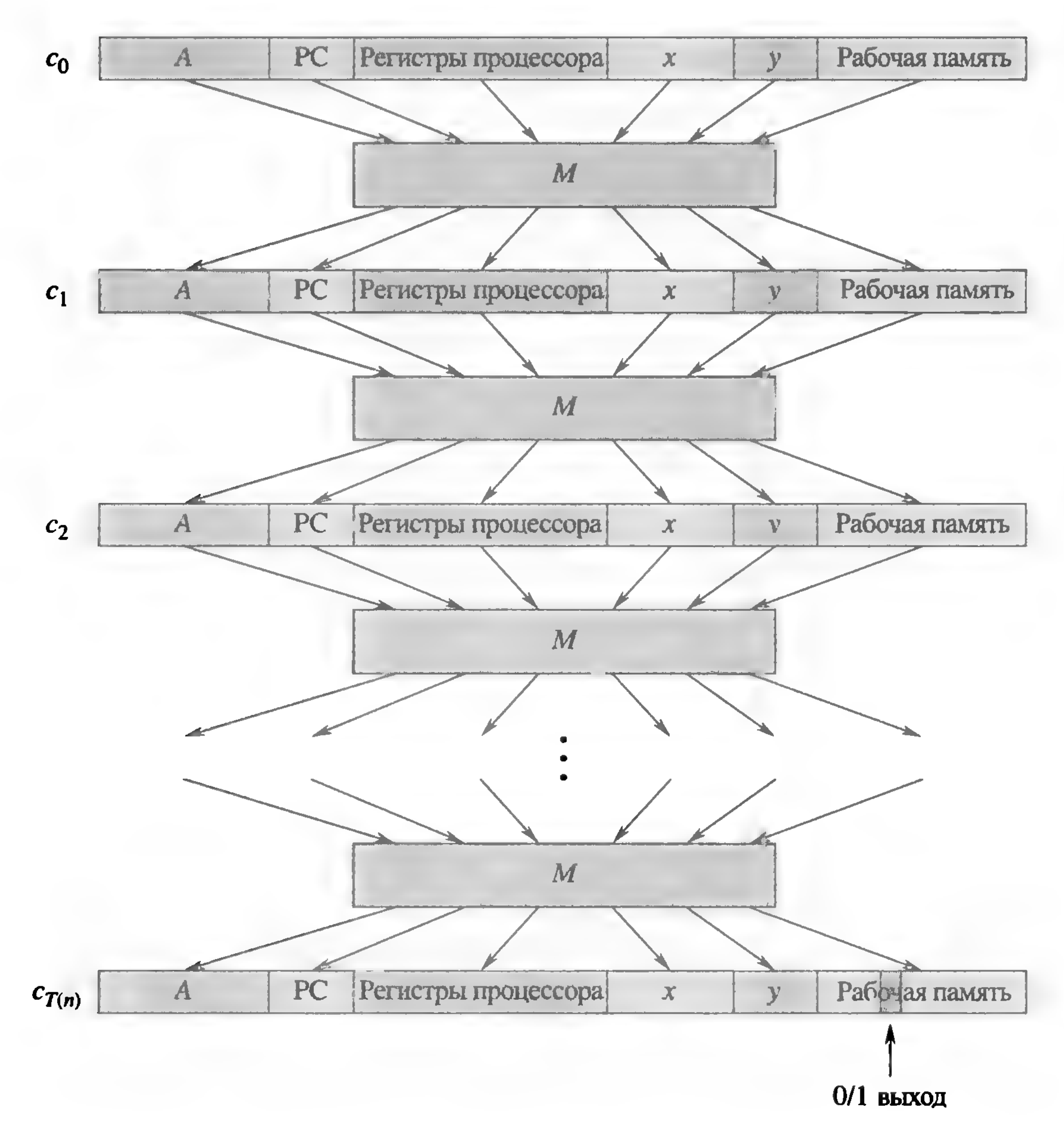


Рис. 34.9. Последовательность конфигураций, полученных в ходе работы алгоритма A, на вход которого поступили строка x и сертификат y. Каждая конфигурация представляет состояние компьютера для одного шага вычислений и, помимо A, x и y, включает счетчик команд (PC), регистры процессора и рабочую память. За исключением сертификата y, исходная конфигурация c_0 является константной. Булева комбинаторная схема M отображает каждую конфигурацию на очередную. Выход представляет собой некоторый предопределенный бит в рабочей памяти.

рис. 34.9, каждую конфигурацию можно разбить на следующие части: программа алгоритма A, счетчик команд PC, регистры процессора, входные данные x, сертификат y и рабочая память. Начиная с исходной конфигурации c_0, каждая конфигурация c_i с помощью комбинационной схемы M, аппаратно реализованной в компьютере, отображается на следующую за ней конфигурацию c_{i+1}. Выходное значение алгоритма A (0 или 1) по завершении выполнения алгоритма A записывается в некоторую специально предназначенную для этого ячейку рабочей памяти. При этом предполагается, что после останова алгоритма A это значение не изменяется. Таким образом, если выполнение алгоритма состоит не более чем из $T(n)$ шагов, результат его работы хранится в определенном бите в $c_{T(n)}$.

Алгоритм приведения F конструирует единую комбинационную схему, вычисляющую все конфигурации, которые получаются из заданной входной конфигурации. Идея заключается в том, чтобы "склеить" вместе все $T(n)$ копий схемы M. Выходные данные i-й схемы, выдающей конфигурацию c_i, подаются непосредственно на вход $(i+1)$-й схемы. Таким образом, эти конфигурации вместо того, чтобы храниться в памяти компьютера, просто передаются по проводам, соединяющим копии схемы M.

Теперь вспомним, что должен делать алгоритм приведения F, время работы которого выражается полиномиальной функцией. Для заданных входных данных x он должен вычислить схему $C = f(x)$, которая была бы выполнима тогда и только тогда, когда существует сертификат y, такой, что $A(x, y) = 1$. Когда алгоритм F получает входное значение x, он сначала вычисляет $n = |x|$ и конструирует комбинационную схему C', состоящую из $T(n)$ копий схемы M. На вход схемы C' подается начальная конфигурация, соответствующая вычислению $A(x, y)$, а выходом этой схемы является конфигурация $c_{T(n)}$.

Схема $C = f(x)$, которую создает алгоритм F, получается путем небольшой модификации схемы C'. Во-первых, входы схемы C', соответствующие программе алгоритма A, начальному значению счетчика команд, входной величине x и начальному состоянию памяти, соединяются проводами непосредственно с этими известными величинами. Таким образом, оставшиеся входы схемы соответствуют сертификату y. Во-вторых, игнорируются все выходы схемы, за исключением одного бита конфигурации $c_{T(n)}$, соответствующего выходному значению алгоритма A. Сконструированная таким образом схема C для любого входного параметра y длиной $O(n^k)$ вычисляет величину $C(y) = A(x, y)$. Алгоритм приведения F, на вход которого подается строка x, вычисляет описанную выше схему C и выдает ее.

Докажем два свойства. Во-первых, покажем, что алгоритм F правильно вычисляет функцию приведения f, т.е. что схема C выполнима тогда и только тогда, когда существует сертификат y, такой, что $A(x, y) = 1$. Во-вторых, покажем, что работа алгоритма F завершается за полиномиальное время.

Чтобы показать, что алгоритм F корректно вычисляет функцию приведения, предположим, что существует сертификат y длиной $O(n^k)$, такой, что $A(x, y) = 1$. Тогда при подаче битов сертификата y на вход схемы C на выходе этой схемы получим значение $C(y) = A(x, y) = 1$. Таким образом, если сертификат существует, то схема C выполнима. Для доказательства в обратном направлении предположим, что схема C выполнима. Тогда существует такое входное значение y, что $C(y) = 1$, откуда можно заключить, что $A(x, y) = 1$. Итак, алгоритм F корректно вычисляет функцию приведения.

Чтобы завершить набросок доказательства, осталось лишь показать, что время работы алгоритма F выражается полиномиальной функцией от $n = |x|$. Первое наблюдение, которое можно сделать, заключается в том, что количество битов, необходимое для представления конфигурации, полиномиально зависит от n. Объем программы самого алгоритма A фиксирован и не зависит от длины его входного параметра x. Длина входного параметра x равна n, а длина сертификата y — $O(n^k)$. Поскольку работа алгоритма состоит не более чем из $O(n^k)$ ша-

гов, объем необходимой ему рабочей памяти также выражается полиномиальной функцией от n. (Предполагается, что эта область памяти непрерывна; в упр. 34.3.5 предлагается обобщить доказательство для случая, когда ячейки, к которым обращается алгоритм A, разбросаны по памяти большего объема, причем разброс ячеек имеет свой вид для каждого входного параметра x.)

Размер комбинационной схемы M, реализующей аппаратную часть компьютера, выражается полиномиальной функцией от длины конфигурации, которая, в свою очередь, является полиномиальной от величины $O(n^k)$ и, следовательно, полиномиально зависит от n. (В большей части такой схемы реализуется логика системы памяти.) Схема C состоит не более чем из $t = O(n^k)$ копий M, поэтому ее размер полиномиально зависит от n. Схему C для входного параметра x можно составить в течение полиномиального времени с помощью алгоритма приведения F, поскольку каждый этап построения длится в течение полиномиального времени. ∎

Таким образом, язык CIRCUIT-SAT не проще любого языка класса NP, а поскольку он относится к классу NP, он является NP-полным.

Теорема 34.7

Задача о выполнимости схем является NP-полной.

Доказательство. Справедливость теоремы непосредственно следует из лемм 34.5 и 34.6, а также из определения NP-полноты. ∎

Упражнения

34.3.1

Убедитесь, что схема, изображенная на рис. 34.8, (б), невыполнима.

34.3.2

Покажите, что отношение $\leq_P$ является транзитивным в отношении языков; другими словами, покажите, что из $L_1 \leq_P L_2$ и $L_2 \leq_P L_3$ следует $L_1 \leq_P L_3$.

34.3.3

Докажите, что $L \leq_P \overline{L}$ тогда и только тогда, когда $\overline{L} \leq_P L$.

34.3.4

Покажите, что в качестве сертификата в альтернативном доказательстве леммы 34.5 можно использовать выполняющий набор. С каким сертификатом легче провести доказательство?

34.3.5

В доказательстве леммы 34.6 предполагается, что рабочая память алгоритма A занимает непрерывную область полиномиального объема. В каком месте доказательства используется это предположение? Покажите, что оно не приводит к потере общности.

34.3.6

Язык L называется ***полным*** (complete) в классе языков C относительно приведения за полиномиальное время, если $L \in C$ и $L' \leq_P L$ для всех $L' \in C$. Покажите, что множества $\emptyset$ и $\{0,1\}^*$ — единственные языки класса P, которые не являются полными в этом классе относительно приведения за полиномиальное время.

34.3.7

Покажите, что по отношению к приведениям за полиномиальное время (см. упр. 34.3.6) L является полным в классе NP тогда и только тогда, когда $\overline{L}$ является полным в co-NP.

34.3.8

Алгоритм приведения F в доказательстве леммы 34.6 строит схему $C = f(x)$ на основе знаний о x, A и k. Профессор заметил, что алгоритм F получает в качестве аргумента только x. В то же время об алгоритме A и константе k, с помощью которой выражается его время работы $O(n^k)$, известно лишь то, что они существуют (поскольку язык L принадлежит к классу NP), но не сами их значения. Из этого профессор делает вывод, что алгоритм F может не суметь построить схему C и что язык CIRCUIT-SAT не обязательно NP-сложный. Объясните, какая ошибка содержится в рассуждениях профессора.

34.4. Доказательства NP-полноты

NP-полнота для задачи о выполнимости схем основана на непосредственном доказательстве соотношения $L \leq_P$ CIRCUIT-SAT для каждого языка $L \in$ NP. В этом разделе будет показано, как доказать NP-полноту языка без непосредственного приведения *каждого* языка из класса NP к заданному языку. Мы проиллюстрируем эту методику, доказав NP-полноту различных задач на выполнимость формул. Намного большее количество примеров применения этой методики содержится в разделе 34.5.

Основой рассматриваемого в этом разделе метода, позволяющего доказать NP-полноту задачи, служит сформулированная ниже лемма.

Лемма 34.8

Если язык L такой, что $L' \leq_P L$ для некоторого $L' \in$ NPC, то L является NP-сложным. Если, кроме того, $L \in$ NP, то $L \in$ NPC.

Доказательство. Поскольку язык L' NP-полный, для всех $L'' \in$ NP мы имеем $L'' \leq_P L'$. Согласно предположению $L' \leq_P L$. Таким образом, из транзитивности (упр. 34.3.2) имеем $L'' \leq_P L$, что показывает, что задача L NP-сложная. Если $L \in$ NP, кроме того, $L \in$ NPC. ■

Другими словами, если язык L', о котором известно, что он — NP-полный, удается свести к языку L, тем самым к этому языку неявно сводится любой язык

класса NP. Таким образом, лемма 34.8 предоставляет метод доказательства NP-полноты языка L, состоящий из перечисленных ниже этапов.

1. Доказывается, что $L \in \mathrm{NP}$.
2. Выбирается язык L', о котором известно, что он NP-полный.
3. Описывается алгоритм, который вычисляет функцию f, отображающую каждый экземпляр $x \in \{0,1\}^*$ языка L' на экземпляр $f(x)$ языка L.
4. Доказывается, что для функции f соотношение $x \in L'$ выполняется тогда и только тогда, когда $f(x) \in L$ для всех $x \in \{0,1\}^*$.
5. Доказывается, что время работы алгоритма, вычисляющего функцию f, полиномиальное.

(На шагах 2–5 доказывается NP-сложность языка L.) Эта методология приведения с помощью одного языка, для которого известно, что он NP-полный, намного проще, чем процесс, когда непосредственно демонстрируется, как выполнить приведение для каждого языка из класса NP. Доказательство соотношения $\mathrm{CIRCUIT\text{-}SAT} \in \mathrm{NPC}$ — первый важный шаг в этом направлении. Знание того факта, что задача о выполнимости схемы является NP-полной, позволяет доказывать NP-полноту других задач намного проще. Более того, по мере расширения списка известных NP-полных задач появляется все больше возможностей для выбора языков, которые будут использоваться для приведения.

Выполнимость формулы

Проиллюстрируем методику приведения, доказав NP-полноту задачи определения, выполнима ли не схема, а формула. Именно для этой задачи впервые была доказана NP-полнота.

Сформулируем задачу о ***выполнимости формулы*** (formula satisfiability) в терминах языка SAT. Экземпляр языка SAT — это булева формула ϕ, состоящая из перечисленных ниже элементов.

1. n булевых переменных: $x_1, x_2, \ldots, x_n$.
2. m булевых соединяющих элементов, в роли которых выступает произвольная логическая функция с одним или двумя входными значениями и одним выходным значением, например $\wedge$ (И), $\vee$ (ИЛИ), $\neg$ (НЕ), $\rightarrow$ (импликация), $\leftrightarrow$ (эквивалентность).
3. Скобки. (Без потери общности предполагается, что лишние скобки не употребляются, т.е. что на каждый логический соединяющий элемент приходится не более одной пары скобок.)

Логическую формулу ϕ легко закодировать строкой, длина которой выражается полиномиальной функцией от $n+m$. Как и для логических комбинационных схем, ***набором значений*** (truth assignment) логической формулы ϕ называется множество значений переменных этой формулы, а ***выполняющим набором*** (satisfying assignment) — такой набор значений, при котором результат вычисления формулы

равен 1. Формула, для которой существует выполняющий набор, является ***выполнимой*** (satisfiable). В задаче о выполнимости спрашивается, выполнима ли данная формула. В терминах формальных языков эта задача имеет вид

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ — выполнимая булева формула}\} \ .$$

В качестве примера приведем формулу

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \ ,$$

у которой имеется выполняющий набор $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, поскольку

$$\begin{aligned} \phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 \ , \end{aligned} \tag{34.2}$$

так что данная формула ϕ принадлежит языку SAT.

Простейший прямолинейный алгоритм, позволяющий определить, выполнима ли произвольная булева формула, не укладывается в полиномиально-временные рамки. В формуле ϕ с n переменными всего имеется 2^n возможных вариантов присваивания. Если длина $\langle\phi\rangle$ выражается полиномиальной функцией от n, то для проверки каждого варианта присваивания потребуется время $\Omega(2^n)$, т.е. оно выражается функцией, показатель роста которой превосходит полиномиальную функцию от длины $\langle\phi\rangle$. Как видно из приведенной ниже теоремы, существование алгоритма с полиномиальным временем маловероятно.

Теорема 34.9

Задача о выполнимости булевых формул является NP-полной.

Доказательство. Сначала докажем, что SAT $\in$ NP. Затем покажем, что язык SAT NP-сложный, для чего продемонстрируем справедливость соотношения CIRCUIT-SAT $\leq_{\text{P}}$ SAT. Согласно лемме 34.8 этого достаточно для доказательства теоремы.

Чтобы показать, что язык SAT относится к классу NP, покажем, что сертификат, состоящий из выполняющего набора входной формулы ϕ, можно верифицировать за полиномиальное время. В алгоритме верификации каждая содержащаяся в формуле переменная просто заменяется соответствующим значением, после чего вычисляется выражение, как это было проделано выше с уравнением (34.2). Эту задачу легко выполнить за полиномиальное время. Если в результате получится значение 1, то формула выполнима. Таким образом, первое условие леммы 34.8 выполняется.

Чтобы доказать, что язык SAT NP-сложный, покажем, что CIRCUIT-SAT $\leq_{\text{P}}$ SAT. Другими словами, покажем, как любой экземпляр задачи о выполнимости схемы можно за полиномиальное время привести к экземпляру задачи о выполнимости формулы. С помощью индукции любую логическую комбинационную

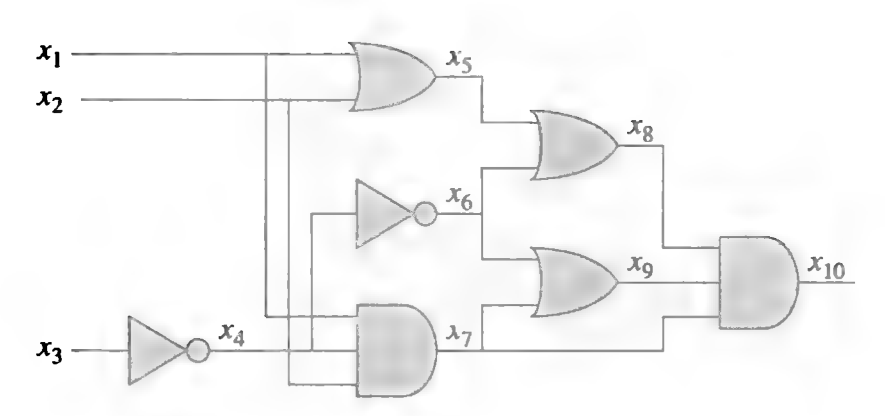


Рис. 34.10. Приведение задачи о выполнимости схем к задаче о выполнимости формул; формула, полученная в результате выполнения алгоритма приведения, содержит переменные, соответствующие каждому проводу схемы.

схему можно выразить в виде булевой формулы. Для этого достаточно рассмотреть элемент, который выдает выходное значение схемы, и по индукции выразить каждое входное значение этого элемента в виде формул. Формула схемы получается путем выписывания выражения, в котором функция элемента применяется к формулам входов.

К сожалению, такой незамысловатый метод не может стать основой приведения за полиномиальное время. Как предлагается показать в упр. 34.4.1, общие вспомогательные формулы, соответствующие элементам, коэффициент ветвления которых равен 2 или превышает это значение, могут привести к экспоненциальному росту размера формулы. Таким образом, алгоритм приведения должен быть несколько остроумнее.

Основная идея приведения задачи CIRCUIT-SAT к задаче SAT для схемы из рис. 34.8, (а) проиллюстрирована на рис. 34.10. Каждому проводу x_i схемы C сопоставляется одноименная переменная формулы ϕ. Тогда надлежащее действие элемента можно выразить в виде небольшой формулы, включающей в себя переменные, которые соответствуют проводам, подсоединенным к этому элементу. Например, действие элемента И выражается формулой $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. Будем называть каждую такую небольшую формулу ***подвыражением*** (clause).

Формула ϕ, которая является результатом выполнения алгоритма приведения, получается путем конъюнкции (элемент И) выходной переменной схемы с выражением, представляющим собой конъюнкцию взятых в скобки подвыражений, описывающих действие каждого элемента. Для схемы, изображенной на рисунке, формула имеет следующий вид:

$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3)\\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge (x_6 \leftrightarrow \neg x_4)\\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\ .\end{aligned}$$

Для заданной схемы C легко получить такую формулу ϕ за полиномиальное время.

Почему схема C выполнима именно тогда, когда выполнима формула ϕ? Если у схемы C имеется выполняющий набор, то по каждому проводу схемы передается вполне определенное значение, и на выходе схемы получается значение 1. Поэтому набор значений, передающихся по проводам схемы, в формуле ϕ приведет к тому, что значение каждого подвыражения в скобках в формуле ϕ будет равно 1, вследствие чего конъюнкция всех этих подвыражений будет равна 1. Верно и обратное: если существует набор данных, для которого значение формулы ϕ равно 1, то схема C выполнима (это можно показать аналогично). Таким образом, показана справедливость соотношения CIRCUIT-SAT $\leq_P$ SAT, что и завершает доказательство теоремы. ■

3-CNF-выполнимость

NP-полноту многих задач можно доказать путем приведения к ним задачи о выполнимости формул. Однако алгоритм приведения должен работать с любыми входными формулами, что может привести к огромному количеству случаев, подлежащих рассмотрению. Поэтому часто желательно выполнять приведение с помощью упрощенной версии языка булевых формул, чтобы уменьшить количество рассматриваемых случаев (конечно же, ограничивать язык настолько, чтобы он стал распознаваемым за полиномиальное время, при этом нельзя). Одним из таких удобных языков является язык формул в 3-конъюнктивной нормальной форме 3-CNF-SAT.

Определим задачу о 3-CNF выполнимости в описанных ниже терминах. ***Литерал*** (literal) в булевой формуле — это входящая в нее переменная или отрицание этой переменной. Булева формула приведена к ***конъюнктивной нормальной форме*** (conjunctive normal form — CNF), если она имеет вид конъюнкции (элемент И) ***выражений*** (clauses), каждое из которых представляет собой дизъюнкцию (элемент ИЛИ) одного или нескольких литералов. Булева формула выражена в ***3-конъюнктивной нормальной форме*** (3-conjunctive normal form), или ***3-CNF***, если в каждом подвыражении в скобках содержится ровно три различных литерала.

Например, булева формула

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

принадлежит классу 3-CNF. Первое из трех подвыражений в скобках имеет вид $(x_1 \vee \neg x_1 \vee \neg x_2)$ и содержит три литерала — x_1, $\neg x_1$ и $\neg x_2$.

В задаче 3-CNF-SAT спрашивается, выполнима ли заданная формула ϕ, принадлежащая классу 3-CNF. В приведенной ниже теореме показано, что алгоритма с полиномиальным временем работы, способного определять выполнимость даже таких простых булевых формул, скорее всего, не существует.

Теорема 34.10

Задача о выполнимости булевых формул в 3-конъюнктивной нормальной форме является NP-полной.

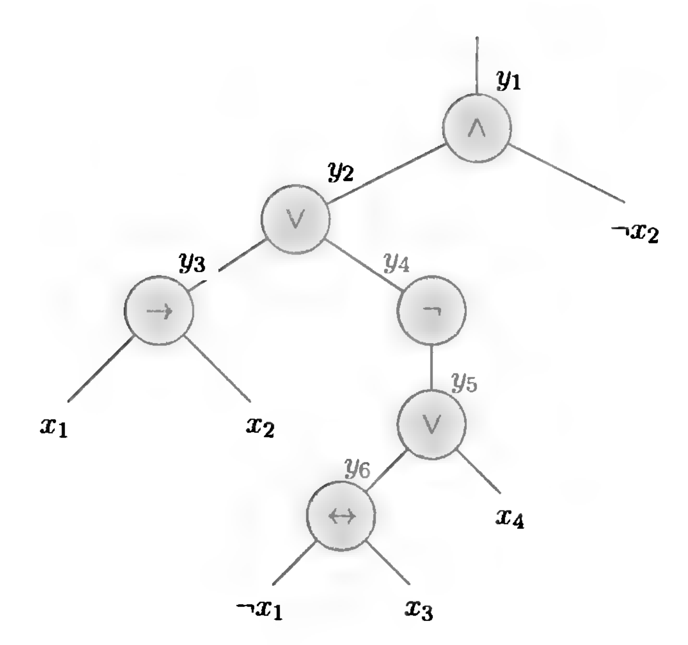


Рис. 34.11. Дерево, соответствующее формуле $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Доказательство. Рассуждения, которые использовались в теореме 34.9 для доказательства того, что SAT $\in$ NP, применимы и для доказательства того, что 3-CNF-SAT $\in$ NP. Таким образом, согласно лемме 34.8 нам требуется лишь показать, что SAT $\leq_P$ 3-CNF-SAT.

Алгоритм приведения можно разбить на три основных этапа. На каждом этапе входная формула ϕ последовательно приводится к 3-конъюнктивной нормальной форме.

Первый этап аналогичен тому, который был использован при доказательстве CIRCUIT-SAT $\leq_P$ SAT в теореме 34.9. Сначала для входной формулы ϕ конструируется бинарное "синтаксическое" дерево, листья которого соответствуют литералам, а узлы — соединительным элементам. На рис. 34.11 показано такое синтаксическое дерево для формулы

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 . \tag{34.3}$$

Если входная формула содержит подвыражение, такое как дизъюнкция или конъюнкция нескольких литералов, то, воспользовавшись свойством ассоциативности, можно провести расстановку скобок в выражении таким образом, чтобы каждый внутренний узел полученного в результате дерева содержал 1 или 2 дочерних узла. Теперь такое бинарное синтаксическое дерево можно рассматривать как схему, предназначенную для вычисления функции.

Имитируя приведение из доказательства теоремы 34.9, введем переменные y_i для выхода из каждого внутреннего узла. Затем перепишем исходную формулу ϕ как конъюнкцию переменной, соответствующей корню дерева, и выражений, описывающих операции в каждом узле. Для формулы (34.3) полученное в результате

выражение имеет следующий вид:

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))\\ &\wedge (y_4 \leftrightarrow \neg y_5)\\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\ .\end{aligned}$$

Обратите внимание, что полученная таким образом формула ϕ' представляет собой конъюнкцию выражений ϕ'_i, каждое из которых содержит не более трех литералов. Единственное требование, которое может оказаться нарушенным, состоит в том, что каждое выражение в скобках должно быть дизъюнкцией трех литералов.

На втором этапе приведения каждое подвыражение в скобках ϕ'_i преобразуется в конъюнктивную нормальную форму. Составим таблицу истинности формулы ϕ'_i путем ее вычисления при всех возможных наборах значений ее переменных. Каждая строка такой таблицы соответствует одному из вариантов набора переменных выражения и содержит сам вариант и результат вычисления исследуемого подвыражения. С помощью элементов таблицы истинности, которые приводят к нулевому результату в формуле, можно записать формулу, эквивалентную подвыражению $\neg\phi'_i$, в ***дизъюнктивной нормальной форме*** (disjunctive normal form — DNF), которая представляет собой дизъюнкцию конъюнкций. Затем эта формула с помощью законов де Моргана

$$\begin{aligned}\neg(a \wedge b) &= \neg a \vee \neg b\ ,\\ \neg(a \vee b) &= \neg a \wedge \neg b\end{aligned}$$

преобразуется в CNF-формулу ϕ''_i путем замены всех литералов их дополнениями и замены операций дизъюнкций и конъюнкций одна другой.

В качестве примера преобразуем выражение $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ в CNF. Таблица истинности для этой формулы представлена на рис. 34.12. DNF-формула, эквивалентная выражению $\neg\phi'_1$, имеет вид

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)\ .$$

Обращая и применяя законы де Моргана, получим CNF-формулу

$$\begin{aligned}\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) &\wedge (\neg y_1 \vee y_2 \vee \neg x_2)\\ \wedge (\neg y_1 \vee y_2 \vee x_2) &\wedge (y_1 \vee \neg y_2 \vee x_2)\ ,\end{aligned}$$

которая эквивалентна исходной формуле ϕ'_1.

Теперь каждое подвыражение ϕ'_i, содержащееся в формуле ϕ', преобразовано в CNF-формулу ϕ''_i, так что ϕ' эквивалентно CNF-формуле ϕ'', состоящей из конъ-

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Рис. 34.12. Таблица истинности для выражения $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

юнкции выражений ϕ_i''. Кроме того, каждое подвыражение формулы ϕ'' содержит не более трех литералов.

На третьем (и последнем) этапе приведения формула преобразуется таким образом, чтобы в каждой паре скобок содержалось *ровно* три различных литерала. Полученная в результате окончательная 3-CNF формула ϕ''' составлена из подвыражений, входящих в CNF-формулу ϕ''. В ней также используются две вспомогательные переменные, которые будут обозначаться как p и q. Для каждого подвыражения C_i из формулы ϕ'' в формулу ϕ''' включаются следующие подвыражения.

- Если C_i содержит три различных литерала, то это подвыражение включается в формулу ϕ''' в неизменном виде.
- Если подвыражение C_i содержит два различных литерала, т.е. если $C_i = (l_1 \vee l_2)$, где l_1 и l_2 — литералы, то в качестве подвыражения в формулу ϕ''' включается выражение $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$. Литералы p и $\neg p$ служат лишь для того, чтобы удовлетворялось требование о наличии в каждом подвыражении в скобках ровно трех разных литералов: когда $p = 0$ или $p = 1$, одно из подвыражений эквивалентно $l_1 \vee l_2$, а второе — единице, что и обеспечивает тождественность первому из упомянутых подвыражений при применении операции И.
- Если подвыражение C_i содержит ровно один литерал l, то вместо него в качестве подвыражения в формулу ϕ''' включается подвыражение $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. При любых значениях переменных p и q одно из четырех подвыражений равно l, а прочие три равны единице.

Проверив каждый из описанных выше трех этапов, можно убедиться, что 3-CNF формула ϕ''' выполнима тогда и только тогда, когда выполнима формула ϕ. Как и в случае приведения задачи CIRCUIT-SAT к задаче SAT, составление на первом этапе формулы ϕ' из формулы ϕ не влияет на выполнимость. На втором этапе конструируется CNF-формула ϕ'', алгебраически эквивалентная формуле ϕ'. На третьем этапе конструируется 3-CNF формула ϕ''', результат которой эквивалентен формуле ϕ'', поскольку присваивание любых значений переменным p и q приводит к формуле, алгебраически эквивалентной формуле ϕ''.

Необходимо также показать, что приведение можно выполнить за полиномиальное время. В ходе конструирования формулы ϕ' по формуле ϕ приходится добавлять не более одной переменной и одного подвыражения на каждый соединительный элемент формулы ϕ. В процессе построения формулы ϕ'' из формулы ϕ' в формулу ϕ'' добавляется не более восьми подвыражений для каждого подвыражения в скобках из формулы ϕ', поскольку каждое подвыражение в этой формуле содержит не более трех переменных и таблица его истинности состоит не более чем из $2^3 = 8$ строк. В ходе конструирования формулы ϕ''' из формулы ϕ'' в формулу ϕ''' добавляется не более четырех подвыражений на каждое подвыражение формулы ϕ''. Таким образом, размер полученной в результате формулы ϕ''' выражается полиномиальной функцией от длины исходной формулы. Следовательно, каждый этап можно легко выполнить за полиномиальное время. ■

Упражнения

34.4.1

Рассмотрите простое приведение "в лоб" (время работы которого не выражается полиномиальной функцией) из теоремы 34.9. Опишите схему размера n, размер которой после преобразования этим методом превратится в формулу, размер которой выражается показательной функцией от n.

34.4.2

Приведите 3-CNF формулу, которая получится в результате применения метода, описанного в теореме 34.10, к формуле (34.3).

34.4.3

Профессор предложил показать, что SAT $\leq_P$ 3-CNF-SAT, лишь с помощью метода с использованием таблицы истинности, описанного в доказательстве теоремы 34.10, исключая при этом другие этапы. Другими словами, профессор предложил взять булеву формулу ϕ, составить таблицу истинности для ее переменных, получить из нее формулу в 3-DNF, эквивалентную $\neg\phi$, после чего составить логическое отрицание полученной формулы и с помощью законов де Моргана получить формулу 3-CNF, эквивалентную формуле ϕ. Покажите, что в результате применения такой стратегии не удается выполнить приведение формулы за полиномиальное время.

34.4.4

Покажите, что задача определения того, является ли тавтологией данная формула, является полной в классе co-NP. (*Указание:* см. упр. 34.3.7).

34.4.5

Покажите, что задача определения выполнимости булевых формул в дизъюнктивной нормальной форме разрешима в течение полиномиального времени.

34.4.6

Предположим, что кто-то разработал алгоритм с полиномиальным временем выполнения, позволяющий решить задачу о выполнимости формул. Опишите, как

с помощью этого алгоритма в течение полиномиального времени находить выполняющие наборы.

34.4.7

Пусть 2-CNF-SAT — множество выполнимых булевых формул в CNF, у которых каждое выражение в скобках содержит ровно по два литерала. Покажите, что 2-CNF-SAT $\in$ P. Постарайтесь, чтобы ваш алгоритм был как можно более эффективным. (*Указание:* воспользуйтесь тем, что выражение $x \vee y$ эквивалентно выражению $\neg x \to y$. Приведите задачу 2-CNF-SAT к задаче на ориентированном графе, имеющей эффективное решение.)

34.5. NP-полные задачи

NP-полные задачи возникают в различных областях: в булевой логике, в теории графов, в арифметике, при разработке сетей, в теории множеств и разбиений, при хранении и поиске информации, при планировании вычислительных процессов, в математическом программировании, в алгебре и теории чисел, при создании игр и головоломок, в теории автоматов и языков, при оптимизации программ, в биологии, в химии, физике и т.п. В настоящем разделе с помощью методики приведения будет доказана NP-полнота различных задач, возникающих в теории графов и при разбиении множеств.

На рис. 34.13 приведена схема доказательства NP-полноты, которая используется в этом разделе и в разделе 34.4. Для каждого из представленных на рисунке языков NP-полнота доказывается путем его приведения к языку, на который указывает идущая от этого языка стрелка. В качестве корневого выступает язык CIRCUIT-SAT, NP-полнота которого доказана в теореме 34.7.

34.5.1. Задача о клике

Клика (clique) неориентированного графа $G = (V, E)$ — это подмножество $V' \subseteq V$ вершин, каждая пара в котором связана ребром из множества E. Другими словами, клика — это полный подграф графа G. ***Размер*** (size) клики — это количество содержащихся в этом подграфе вершин. ***Задача о клике*** (clique problem) — это задача оптимизации, в которой требуется найти клику максимального размера, содержащуюся в заданном графе. В соответствующей задаче принятия решения спрашивается, содержится ли в графе клика заданного размера k. Формальное определение языка имеет вид

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ — граф, содержащий клику размером } k\} .$$

Простейший алгоритм определения того, содержит ли граф $G = (V, E)$ с $|V|$ вершинами клику размером k, заключается в том, чтобы перечислить все k-элементные подмножества множества V и проверить, образует ли каждое из них клику. Время работы этого алгоритма равно $\Omega(k^2\binom{|V|}{k})$ и является полиномиаль-

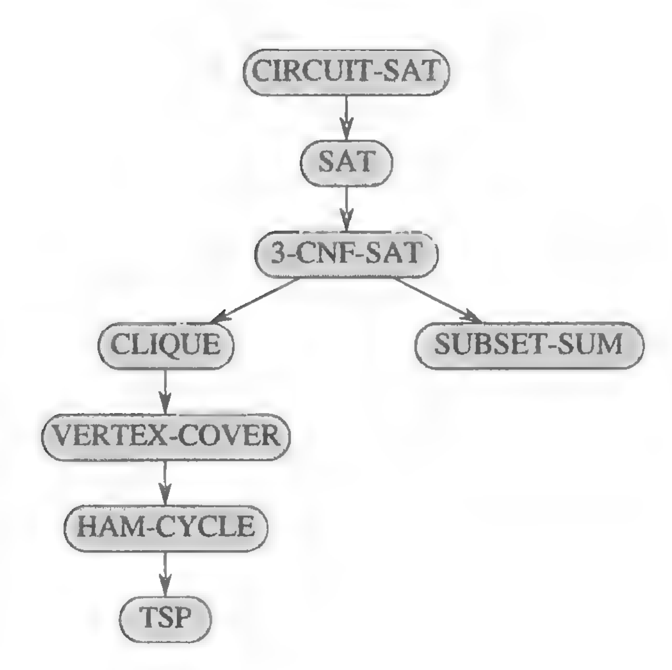


Рис. 34.13. Схема доказательств NP-полноты задач, рассматриваемых в разделах 34.4 и 34.5. Все доказательства в конечном итоге сводятся к приведению NP-полной задачи CIRCUIT-SAT к рассматриваемой.

ным, если k — константа. Однако в общем случае величина k может достигать значения, близкого к $|V|/2$, и в этом случае время работы алгоритма превышает полиномиальное. Есть основания предполагать, что эффективного алгоритма для задачи о клике не существует.

Теорема 34.11

Задача о клике является NP-полной.

Доказательство. Чтобы показать, что CLIQUE $\in$ NP, для заданного графа $G = (V, E)$, воспользуемся множеством $V' \subseteq V$ вершин клики в качестве сертификата для данного графа. Проверить, является ли множество вершин V' кликой, можно в течение полиномиального времени, проверяя для каждой пары вершин $u, v \in V'$, принадлежит ли соединяющее их ребро (u, v) множеству E.

Теперь докажем, что 3-CNF-SAT $\leq_P$ CLIQUE, откуда следует, что задача о клике является NP-сложной. То, что этот результат удается доказать, может показаться удивительным, потому что, на первый взгляд, логические формулы мало напоминают графы.

Алгоритм приведения начинается с экземпляра задачи 3-CNF-SAT. Пусть $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ — булева формула из класса 3-CNF с k подвыражениями. Каждое подвыражение C_r при $r = 1, 2, \ldots, k$ содержит ровно три различных литерала — l_1^r, l_2^r и l_3^r. Сконструируем такой граф G, чтобы формула ϕ была выполнима тогда и только тогда, когда граф G содержит клику размера k.

Граф $G = (V, E)$ мы строим следующим образом. Для каждого подвыражения $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ в ϕ помещаем тройку вершин, v_1^r, v_2^r и v_3^r, в V. Вершины v_i^r и v_j^s соединяются ребром, если справедливы оба следующих утверждения:

- v_i^r и v_j^s находятся в разных тройках, т.е. $r \neq s$; и
- их литералы ***совместимы***, т.е. l_i^r не является отрицанием l_j^s.

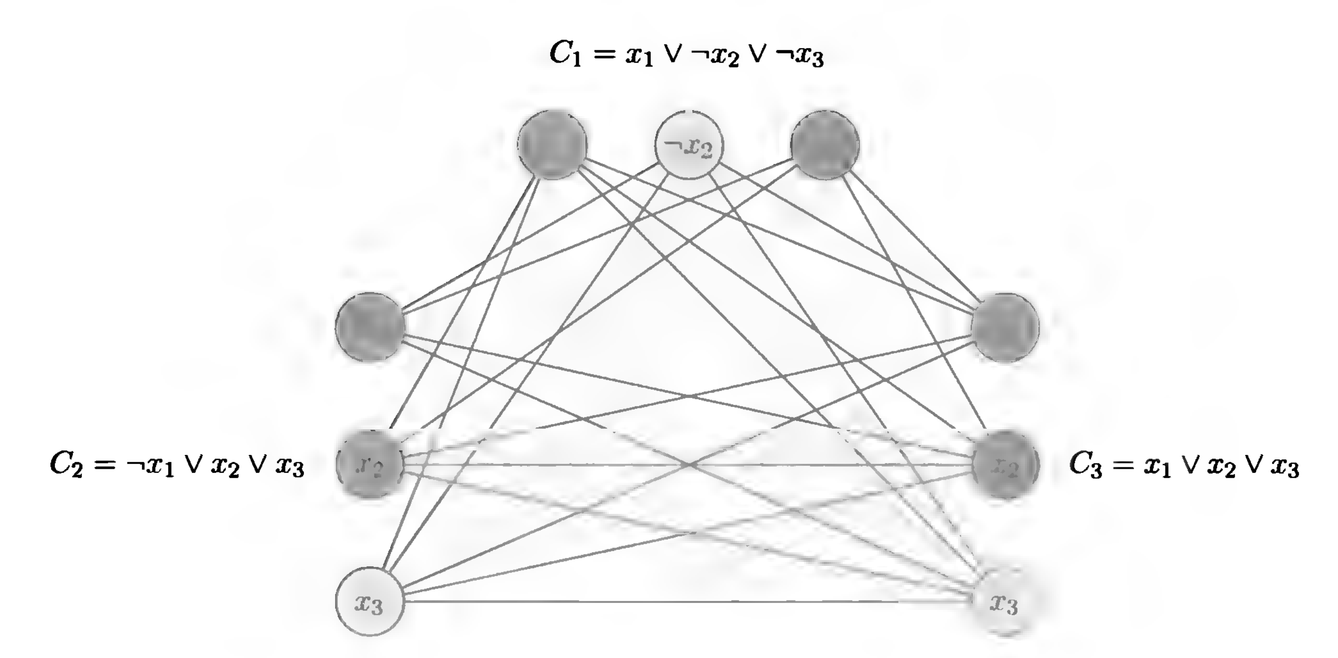


Рис. 34.14. Граф G, полученный в процессе приведения 3-CNF-SAT к задаче CLIQUE из 3-CNF-формулы $\phi = C_1 \wedge C_2 \wedge C_3$, где $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ и $C_3 = (x_1 \vee x_2 \vee x_3)$. Выполняющий набор формулы включает $x_2 = 0$, $x_3 = 1$, а x_1 может быть как нулем, так и единицей. Этот набор выполняет C_1 благодаря $\neg x_2$, и C_2 и C_3 благодаря x_3, соответствуя клике из выделенных светлой штриховкой вершин.

Такой граф легко построить для формулы ϕ за полиномиальное время. В качестве примера на рис. 34.14 приведен граф G, соответствующий формуле

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) .$$

Необходимо показать, что это преобразование формулы ϕ в граф G является приведением. Во-первых, предположим, что для формулы ϕ имеется выполняющий набор. Тогда каждое подвыражение C_r содержит не менее одного литерала l_i^r, значение которого равно единице, и каждый такой литерал соответствует вершине v_i^r. В результате извлечения такого "истинного" литерала из каждого подвыражения получается множество V', состоящее из k вершин. Мы утверждаем, что V' — клика. Для любых двух вершин $v_i^r, v_j^s \in V'$, где $r \neq s$, оба соответствующих литерала, l_i^r и l_j^s, при данном выполняющем наборе равны единице, поэтому они не могут быть отрицаниями один другого. Таким образом, в соответствии со способом построения графа G ребро (v_i^r, v_j^s) принадлежит множеству E.

Проведем обратные рассуждения. Предположим, что граф G содержит клику V' размером k. Ни одно из ее ребер не соединяет вершины одной и той же тройки, поэтому клика V' содержит ровно по одной вершине каждой тройки. Каждому литералу l_i^r, такому, что $v_i^r \in V'$, можно присвоить значение 1, не опасаясь того, что оно будет присвоено как литералу, так и его дополнению, поскольку граф G не содержит ребер, соединяющих противоречивые литералы. Каждое подвыражение является выполнимым, поэтому выполнима и формула ϕ. (Переменным, которым не соответствует ни одна вершина клики, можно присваивать произвольные значения.) ■

В примере, представленном на рис. 34.14, выполняющий набор для формулы ϕ имеет вид $x_2 = 0$ и $x_3 = 1$. Соответствующая клика размером $k = 3$ состоит из вершин, отвечающих литералу $\neg x_2$ из первого подвыражения в скобках, литералу x_3 из второго выражения в скобках и литералу x_3 из третьего выражения в скобках. Поскольку клика не содержит вершин, соответствующих литералу x_1 или литералу $\neg x_1$, переменная x_1 в выполняющем наборе может принимать как значение 0, так и значение 1.

Заметим, что при доказательстве теоремы 34.11 произвольный экземпляр задачи 3-CNF-SAT был сведен к экземпляру задачи CLIQUE, обладающему определенной структурой. Может показаться, что принадлежность задачи CLIQUE категории NP-сложных была доказана лишь для графов, все вершины которых можно разбить на тройки, причем таких, в которых отсутствуют ребра, соединяющие вершины одной и той же тройки. В самом деле, NP-сложность задачи CLIQUE была показана только для этого ограниченного случая, однако этого доказательства достаточно, чтобы сделать вывод о NP-сложности этой задачи для графа общего вида. Почему? Дело в том, что из наличия алгоритма с полиномиальным временем работы, решающего задачу CLIQUE с графами общего вида, следует существование такого алгоритма решения этой задачи с графами, имеющими ограниченную структуру.

Однако обратный подход — приведение экземпляров задачи 3-CNF-SAT, обладающих какой-то особой структурой, к экземплярам задачи CLIQUE общего вида, был бы недостаточен. Почему? Может случиться так, что экземпляры задачи 3-CNF-SAT, с помощью которых выполняется приведение, окажутся слишком "легкими", и к задаче CLIQUE приводится задача, не являющаяся NP-сложной.

Заметим также, что для приведения используется экземпляр задачи 3-CNF-SAT, но не ее решение. Было бы ошибкой основывать приведение за полиномиальное время на знании того, выполнима ли формула ϕ, поскольку неизвестно, как получить эту информацию за полиномиальное время.

34.5.2. Задача о вершинном покрытии

Вершинное покрытие (vertex cover) неориентированного графа $G = (V, E)$ — это такое подмножество $V' \subseteq V$, что если $(u, v) \in E$, то либо $u \in V'$, либо $v \in V'$, либо справедливы оба эти соотношения. Другими словами, каждая вершина "покрывает" инцидентные ребра, а вершинное покрытие графа G — это множество вершин, покрывающих все ребра из множества E. ***Размером*** (size) вершинного покрытия называется количество содержащихся в нем вершин. Например, граф, изображенный на рис. 34.15, (б), имеет вершинное покрытие $\{w, z\}$ размером 2.

Задача о вершинном покрытии (vertex-cover problem) заключается в том, чтобы найти в заданном графе вершинное покрытие минимального размера. Переформулируем эту задачу оптимизации в виде задачи принятия решения, в которой требуется определить, содержит ли граф вершинное покрытие заданного размера k. Определим язык

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{граф } G \text{ имеет вершинное покрытие размером } k\} .$$

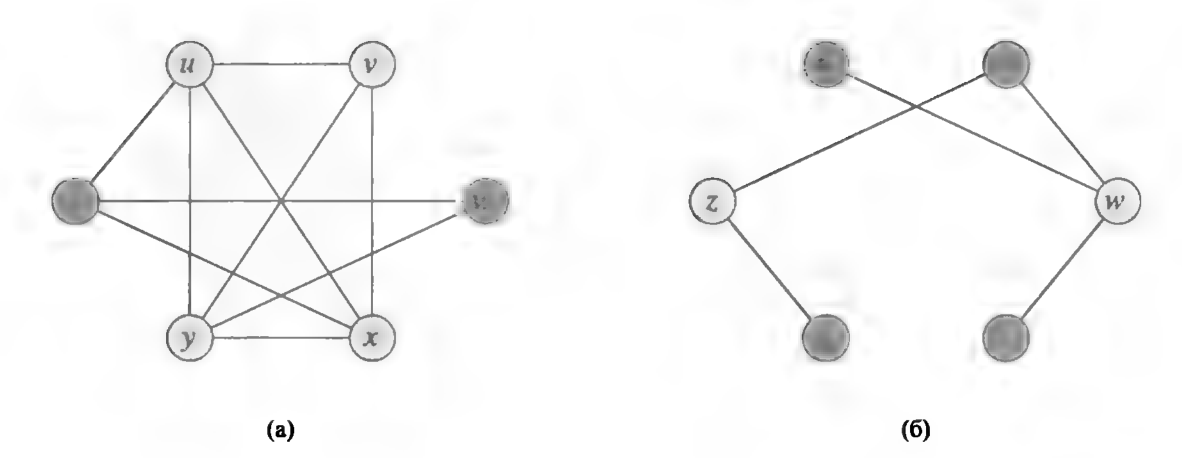


Рис. 34.15. Приведение CLIQUE к VERTEX-COVER. **(а)** Неориентированный граф $G = (V, E)$ с кликой $V' = \{u, v, x, y\}$. **(б)** Полученный алгоритмом приведения граф $\overline{G}$, обладающий вершинным покрытием $V - V' = \{w, z\}$.

В сформулированной ниже теореме доказывается, что эта задача является NP-полной.

Теорема 34.12

Задача о вершинном покрытии является NP-полной.

Доказательство. Сначала покажем, что VERTEX-COVER $\in$ NP. Предположим, что заданы граф $G = (V, E)$ и целое число k. В качестве сертификата выберем само вершинное покрытие $V' \subseteq V$. В алгоритме верификации проверяется, что $|V'| = k$, а затем для каждого ребра $(u, v) \in E$ проверяется, что $u \in V'$ или $v \in V'$. Такую верификацию можно выполнить непосредственно, как описано выше, за полиномиальное время.

Докажем, что задача о вершинном покрытии NP-сложная, для чего покажем справедливость соотношения CLIQUE $\leq_{\text{P}}$ VERTEX-COVER. Это приведение основывается на понятии "дополнения" графа. ***Дополнение*** (complement) данного неориентированного графа $G = (V, E)$ определяется как $\overline{G} = (V, \overline{E})$, где $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ и } (u, v) \notin E\}$. Другими словами, $\overline{G}$ — это граф, содержащий те ребра, которых нет в графе G. На рис. 34.15 показаны граф и его дополнение, а также проиллюстрировано приведение задачи CLIQUE к задаче VERTEX-COVER.

Алгоритм приведения в качестве входных данных получает экземпляр $\langle G, k \rangle$ задачи о клике. В этом алгоритме вычисляется дополнение $\overline{G}$, что легко осуществить за полиномиальное время. Выходом алгоритма приведения является экземпляр $\langle \overline{G}, |V| - k \rangle$ задачи о вершинном покрытии. Чтобы завершить доказательство, покажем, что это преобразование действительно является приведением: граф G содержит клику размера k тогда и только тогда, когда граф $\overline{G}$ имеет вершинное покрытие размером $|V| - k$.

Предположим, что граф G содержит клику $V' \subseteq V$ размером $|V'| = k$. Мы утверждаем, что $V - V'$ — вершинное покрытие графа $\overline{G}$. Пусть (u, v) — произвольное ребро из множества $\overline{E}$. Тогда $(u, v) \notin E$, из чего следует, что хотя бы одна из вершин u и v не принадлежит множеству V', поскольку каждая пара вершин из V' соединена ребром, входящим в множество E. Эквивалентно хотя бы одна

из вершин u и v принадлежит множеству $V - V'$, а следовательно, ребро (u, v) покрывается этим множеством. Поскольку ребро (u, v) выбрано из множества $\overline{E}$ произвольным образом, каждое ребро из этого множества покрывается вершиной из множества $V - V'$. Таким образом, множество $V - V'$, размер которого — $|V| - k$, образует вершинное покрытие графа $\overline{G}$.

Справедливо и обратное. Предположим, что граф $\overline{G}$ имеет вершинное покрытие $V' \subseteq V$, где $|V'| = |V| - k$. Тогда для всех пар вершин $u, v \in V$ из $(u, v) \in \overline{E}$ следует, что или $u \in V'$, или $v \in V'$, или справедливы оба эти утверждения. Обращение следствия дает, что для всех пар вершин $u, v \in V$, если $u \notin V'$ и $v \notin V'$, то $(u, v) \in E$. Другими словами, $V - V'$ — это клика, а ее размер равен $|V| - |V'| = k$. ■

Поскольку задача VERTEX-COVER является NP-полной, маловероятным представляется то, что удастся разработать алгоритм поиска вершинного покрытия минимального размера за полиномиальное время. Однако в разделе 35.1 представлен "приближенный алгоритм" с полиномиальным временем работы, позволяющий находить "приближенные" решения этой задачи. Размер вершинного покрытия, полученного в результате работы этого алгоритма, не более чем в два раза превышает размер минимального вершинного покрытия.

Таким образом, не стоит лишать себя надежды только из-за того, что задача NP-полная. Может оказаться, что для нее существует приближенный алгоритм с полиномиальным временем работы, позволяющий получать решения, близкие к оптимальным. В главе 35 описано несколько приближенных алгоритмов, предназначенных для решения NP-полных задач.

34.5.3. Задача о гамильтоновых циклах

Вновь вернемся к задаче о гамильтоновых циклах, определенной в разделе 34.2.

Теорема 34.13

Задача о гамильтоновых циклах является NP-полной.

Доказательство. Сначала покажем, что задача HAM-CYCLE принадлежит классу NP. Для заданного графа $G = (V, E)$ сертификат задачи имеет вид последовательности, состоящей из $|V|$ вершин, образующих гамильтонов цикл. В алгоритме верификации проверяется, что в эту последовательность каждая вершина из V входит ровно по одному разу и что, если повторить первую вершину после последней, образуется цикл в графе G. Другими словами, проверяется, что каждая пара последовательных вершин соединена ребром, а также что ребро соединяет первую и последнюю вершины последовательности. Подобную проверку можно выполнить за полиномиальное время.

Докажем теперь, что VERTEX-COVER $\leq_P$ HAM-CYCLE, откуда следует, что задача HAM-CYCLE NP-полная. Построим для заданного неориентированного графа $G = (V, E)$ и целого числа k неориентированный граф $G' = (V', E')$,

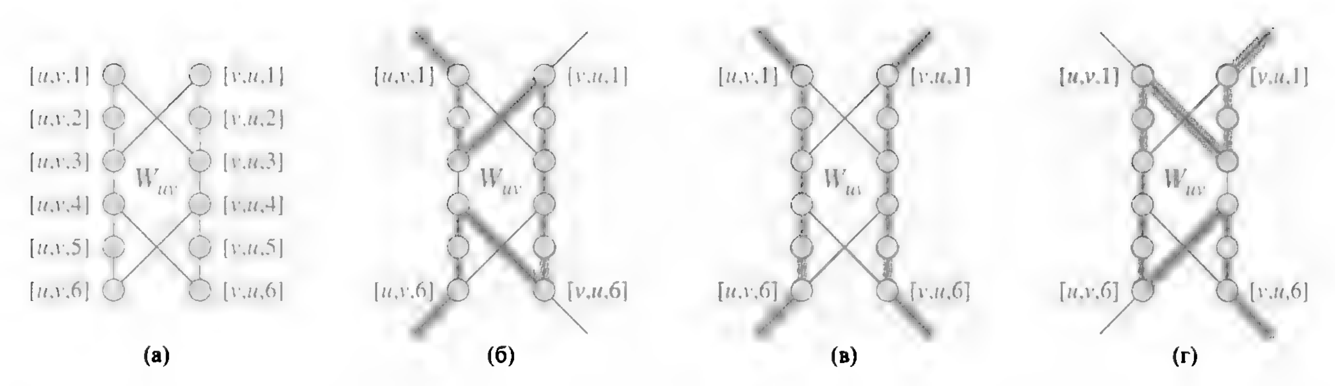


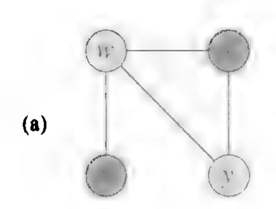
Рис. 34.16. Структурный элемент графа, используемый в ходе приведения задачи о вершинном покрытии к задаче о гамильтоновых циклах. Ребро (u, v) графа G соответствует структурному элементу W_{uv} в графе G', создаваемом в процессе приведения. **(а)** Структурный элемент с помеченными отдельными вершинами. **(б)–(г)** Выделенные штриховкой пути являются единственными возможными путями через структурный элемент, включающими все его вершины, в предположении, что структурные элементы соединены с остальной частью графа G' вершинами $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ и $[v, u, 6]$.

в котором гамильтонов цикл содержится тогда и только тогда, когда размер вершинного покрытия графа G равен k.

Наше построение основывается на ***структурном элементе*** (widget), представляющем собой фрагмент графа, обеспечивающий его определенные свойства. На рис. 34.16, (а) показан используемый нами структурный элемент. Для каждого ребра $(u, v) \in E$ строящийся граф G' будет содержать одну копию этого структурного элемента, обозначаемую как W_{uv}. Обозначим каждую вершину структурного элемента W_{uv} как $[u, v, i]$ или $[v, u, i]$, где $1 \leq i \leq 6$, так что каждый структурный элемент W_{uv} содержит 12 вершин. Кроме того, он содержит 14 ребер, как показано на рис. 34.16, (а).

Требуемые свойства графа обеспечиваются не только внутренней структурой описанного выше элемента, но и наложением ограничений на связи между структурным элементом и остальной частью строящегося графа G'. В частности, наружу структурного элемента W_{uv} будут выходить ребра только из вершин $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ и $[v, u, 6]$. Любой гамильтонов цикл G' должен проходить по ребрам структурного элемента W_{uv} одним из трех способов, показанных на рис. 34.16, (б)–(г). Если цикл входит через вершину $[u, v, 1]$, выйти он должен через вершину $[u, v, 6]$, и при этом он либо проходит через все 12 вершин структурного элемента (рис. 34.16, (б)), либо только через 6 его вершин, с $[u, v, 1]$ по $[u, v, 6]$ (рис. 34.16, (в)). В последнем случае цикл должен будет повторно войти в структурный элемент, посещая вершины с $[v, u, 1]$ по $[v, u, 6]$. Аналогично, если цикл входит в структурный элемент через вершину $[v, u, 1]$, он должен выйти из вершины $[v, u, 6]$, посетив на своем пути либо все 12 вершин (рис. 34.16, (г)), либо 6 вершин, начиная с вершины $[v, u, 1]$ и заканчивая вершиной $[v. u, 6]$ (рис. 34.16, (в)). Никакие другие варианты прохождения всех 12 вершин структурного элемента невозможны. В частности, невозможно таким образом построить два непересекающихся пути, один из которых соединяет вершины $[u, v, 1]$ и $[v, u, 6]$, а другой – вершины $[v, u, 1]$ и $[u, v, 6]$ так, чтобы объединение этих путей содержало все вершины структурного элемента.

(а)



(б)

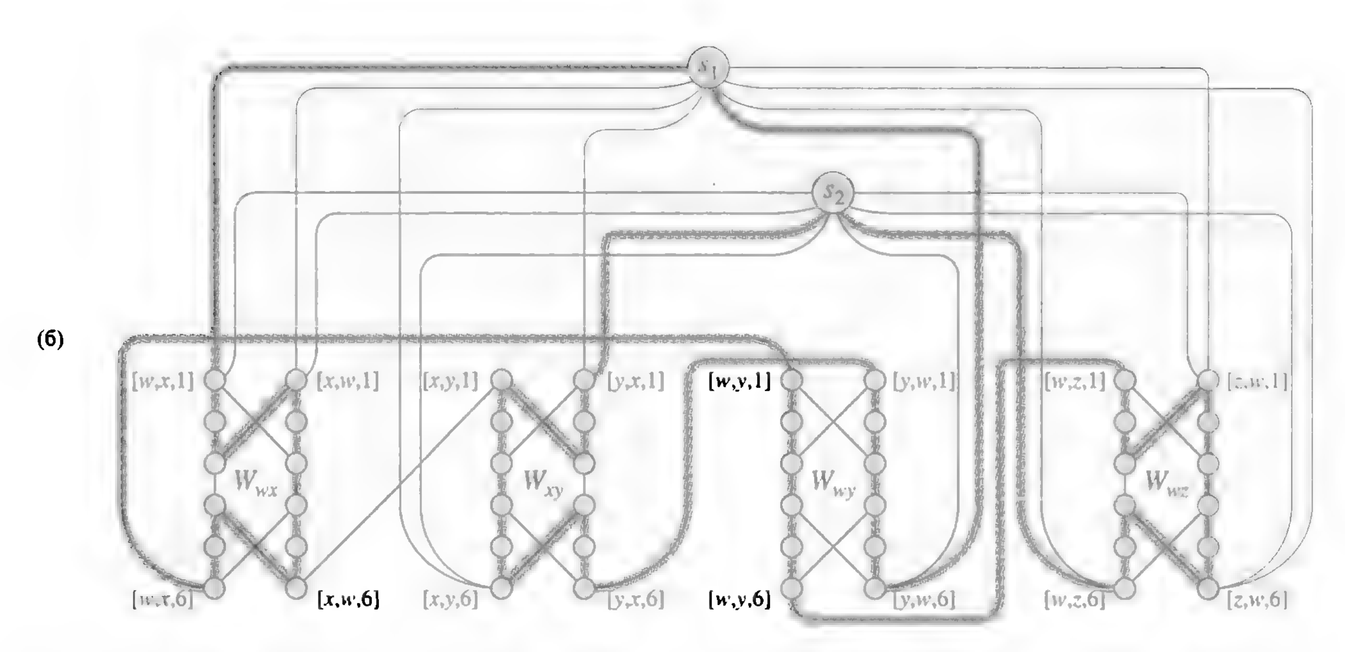


Рис. 34.17. Приведение экземпляра задачи о вершинном покрытии к экземпляру задачи о гамильтоновом цикле. **(а)** Неориентированный граф G с вершинным покрытием размером 2, состоящим из выделенных светлой штриховкой вершин w и y. **(б)** Неориентированный граф G', полученный путем приведения, с выделенным гамильтоновым путем. Вершинное покрытие $\{w, y\}$ соответствует ребрам $(s_1, [w, x, 1])$ и $(s_2, [y, x, 1])$, встречающимся в гамильтоновом цикле.

Единственные вершины, содержащиеся в множестве V', кроме вершин структурных элементов, — ***переключающие вершины*** (selector vertex) $s_1, s_2, \ldots, s_k$. Ребра графа G', инцидентные переключающим вершинам, используются для выбора k вершин из покрытия графа G.

В дополнение к ребрам, входящим в состав структурных элементов, множество E' содержит ребра двух других типов, показанных на рис. 34.17. Во-первых, для каждой вершины $u \in V$ добавляются ребра, соединяющие пары структурных элементов в таком порядке, чтобы получился путь, содержащий все структурные элементы, соответствующие ребрам, инцидентным вершине u графа G. Вершины, смежные с каждой вершиной $u \in V$, упорядочиваются произвольным образом как $u^{(1)}, u^{(2)}, \ldots, u^{(\text{degree}(u))}$, где $\text{degree}(u)$ — количество вершин, смежных с вершиной u. В графе G' создается путь, проходящий через все структурные элементы, соответствующие инцидентным ребрам вершины u. Для этого к множеству E' добавляются ребра $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \leq i \leq \text{degree}(u) - 1\}$. Например, на рис. 34.17 вершины, смежные с вершиной w, упорядочиваются как x, y, z, так что граф G', изображенный в части (б) рисунка, содержит ребра $([w, x, 6], [w, y, 1])$ и $([w, y, 6], [w, z, 1])$. Для каждой вершины $u \in V$ эти ребра графа G' заполняют путь, содержащий все структурные элементы, соответствующие ребрам, инцидентным вершине u графа G.

Интуитивные соображения, обосновывающие наличие этих ребер, заключаются в том, что если из вершинного покрытия графа G выбирается вершина $u \in V$, то в графе G' можно построить путь из вершины $[u, u^{(1)}, 1]$ в вершину $[u, u^{(\text{degree}(u))}, 6]$, "покрывающий" все структурные элементы, соответствующие ребрам, инцидентным вершине u. Другими словами, для каждого из этих структурных элементов, скажем, структурного элемента $W_{u,u^{(i)}}$, этот путь проходит либо по всем 12 вершинам (если вершина u входит в вершинное покрытие, а вершина $u^{(i)}$ — нет), либо только по 6 вершинам $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \ldots, [u, u^{(i)}, 6]$ (если вершинному покрытию принадлежат и вершина u, и вершина $u^{(i)}$).

Наконец последний тип ребер множества E' соединяет первую $[u, u^{(1)}, 1]$ и последнюю $[u, u^{(\text{degree}(u))}, 6]$ вершины каждого из этих путей с каждой из переключающих вершин. Другими словами, в множество включаются ребра

$$\begin{aligned}&\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ и } 1 \leq j \leq k\} \\ &\qquad \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ и } 1 \leq j \leq k\} .\end{aligned}$$

Теперь покажем, что размер графа G' выражается полиномиальной функцией от размера графа G, поэтому граф G' можно сконструировать за полиномиальное от размера графа G время. Множество вершин графа G' состоит из вершин, входящих в состав структурных элементов, и переключающих вершин. Каждый структурный элемент содержит 12 вершин, и еще имеется $k \leq |V|$ переключающих вершин, поэтому в итоге получается

$$\begin{aligned}|V'| &= 12\,|E| + k \\ &\leq 12\,|E| + |V|\end{aligned}$$

вершин. Множество ребер графа G' состоит из ребер, которые принадлежат структурным элементам, ребер, которые соединяют структурные элементы, и ребер, которые соединяют переключающие вершины со структурными элементами. Каждый структурный элемент содержит по 14 ребер, поэтому все структурные элементы в совокупности содержат $14\,|E|$ ребер. Для каждой вершины $u \in V$ граф G' содержит $\text{degree}(u) - 1$ ребер между структурными элементами, так что в результате суммирования по всем вершинам множества V получается

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2\,|E| - |V|$$

ребер, соединяющих структурные элементы. Наконец по два ребра приходится на каждую пару, состоящую из одной переключающей вершины и одной вершины множества V. Всего таких ребер получается $2k\,|V|$, а общее количество всех ребер графа G' равно

$$\begin{aligned}|E'| &= (14\,|E|) + (2\,|E| - |V|) + (2k\,|V|) \\ &= 16\,|E| + (2k - 1)\,|V| \\ &\leq 16\,|E| + (2\,|V| - 1)\,|V| .\end{aligned}$$

Теперь покажем, что преобразование графа G в граф G' является приведением. Другими словами, покажем, что граф G имеет вершинное покрытие размером k тогда и только тогда, когда граф G' имеет гамильтонов цикл.

Предположим, что граф $G = (V, E)$ содержит вершинное покрытие $V^* \subseteq V$ размером k. Пусть $V^* = \{u_1, u_2, \ldots, u_k\}$. Как видно из рис. 34.17, гамильтонов цикл графа G' образуется путем включения в него следующих ребер[10] для каждой вершины $u_j \in V^*$. В цикл включаются ребра $\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u_j) - 1\}$, которые соединяют все структурные элементы, соответствующие ребрам, инцидентным вершинам u_j. Включаются также ребра, содержащиеся в этих структурных элементах, как показано на рис. 34.16, (б)–(г), в зависимости от того, покрывается ребро одним или двумя вершинами множества V^*. Гамильтонов цикл также включает в себя ребра

$$\begin{aligned}\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \le j \le k\}& \\ \cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \le j \le k-1\}& \\ \cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\}& .\end{aligned}$$

Ознакомившись с рис. 34.17, можно убедиться, что эти ребра действительно образуют цикл. Этот цикл начинается в вершине s_1, проходит через все структурные элементы, соответствующие ребрам, инцидентным вершине u_1, затем направляется к вершине u_2, проходит через все структурные элементы, соответствующие ребрам, инцидентным вершине u_2, и так до тех пор, пока снова не вернется к вершине s_1. Каждый структурный элемент проходится однократно или дважды в зависимости от того, одна или две вершины множества V^* покрывают соответствующее ему ребро. Поскольку V^* — вершинное покрытие графа G, каждое ребро из множества E инцидентно некоторой вершине множества V^*, поэтому цикл проходит через все вершины каждого структурного элемента графа G'. Поскольку он также проходит через все переключающие вершины, этот цикл — гамильтонов.

Проведем рассуждения в обратном направлении. Предположим, что граф $G' = (V', E')$ содержит гамильтонов цикл $C \subseteq E'$. Мы утверждаем, что множество

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ для некоторого } 1 \le j \le k\} \tag{34.4}$$

является вершинным покрытием графа G. Чтобы убедиться, что это действительно так, разобьем цикл C на максимальные пути, которые начинаются в некоторой переключающей вершине s_i, проходят через ребро $(s_i, [u, u^{(1)}, 1])$ для некоторой вершины $u \in V$ и оканчиваются в переключающей вершине s_j, не пересекая при этом никакие другие переключающие вершины. Назовем каждый такой путь "покрывающим". По способу построения графа G' каждый покрывающий путь должен начинаться в некоторой вершине s_i, включать в себя ребро $(s_i, [u, u^{(1)}, 1])$

[10]Технически определение цикла формулируется в терминах вершин, а не ребер (см. раздел Б.4). Для ясности здесь эти обозначения видоизменяются, а гамильтонов цикл определяется в терминах ребер.

для некоторой вершины $u \in V$, проходить через все структурные элементы, соответствующие ребрам из множества E, инцидентным вершине u, и оканчиваться в некоторой переключающей вершине s_j. Обозначим такой покрывающий путь как p_u и в соответствии с уравнением (34.4) включим вершину u в множество V^*. Каждый структурный элемент, через который проходит путь p_u, должен быть структурным элементом W_{uv} или структурным элементом W_{vu} для некоторой вершины $v \in V$. О каждом структурном элементе, через который проходит путь p_u, можно сказать, что по его вершинам проходит один или два покрывающих пути. Если такой покрывающий путь один, то ребро $(u, v) \in E$ покрывается в графе G вершиной u. Если же через структурный элемент проходят два пути, то один из них, очевидно, путь p_u, а другой должен быть путем p_v. Из этого следует, что $v \in V^*$, и ребро $(u, v) \in E$ покрывается и вершиной u, и вершиной v. Поскольку все вершины из каждого структурного элемента посещаются некоторым покрывающим путем, видно, что каждое ребро из множества E покрывается некоторой вершиной из множества V^*. ∎

34.5.4. Задача о коммивояжере

В ***задаче о коммивояжере*** (traveling-salesman problem), которая тесно связана с задачей о гамильтоновом цикле, коммивояжер должен посетить n городов. Моделируя задачу в виде полного графа с n вершинами, можно сказать, что коммивояжеру нужно совершить ***тур*** (tour), или гамильтонов цикл, посетив каждый город ровно по одному разу и завершив путешествие в том же городе, из которого он выехал. С каждым переездом из города i в город j связана некоторая стоимость $c(i, j)$, выражаемая целым числом, и коммивояжеру нужно совершить тур таким образом, чтобы общая стоимость (т.е. сумма стоимостей всех переездов) была минимальной. Например, на рис. 34.18 изображен самый дешевый тур $\langle u, w, v, x, u\rangle$, стоимость которого равна 7. Вот как выглядит формальный язык для соответствующей задачи принятия решения:

$$\begin{aligned} \text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ — полный граф}, \\ & c \text{ — функция } V \times V \to \mathbb{N}, \\ & k \in \mathbb{N}, \text{ и} \\ & G \text{ содержит тур стоимостью не более } k\} \ . \end{aligned}$$

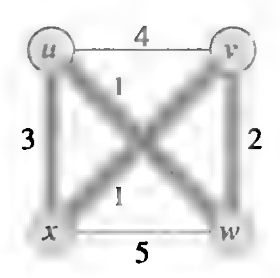


Рис. 34.18. Экземпляр задачи о коммивояжере; выделенные серым цветом ребра представляют самый дешевый тур, стоимость которого равна 7.

Согласно сформулированной ниже теореме существование быстрого алгоритма для решения задачи о коммивояжере маловероятно.

Теорема 34.14

Задача о коммивояжере является NP-полной.

Доказательство. Сначала докажем, что TSP принадлежит классу NP. В качестве сертификата для заданного экземпляра задачи будет использоваться последовательность n вершин, из которых состоит тур. В алгоритме верификации проверяется, что в этой последовательности все вершины содержатся ровно по одному разу, а также суммируются стоимости всех ребер тура и проверяется, что эта сумма не превышает k. Очевидно, что этот процесс можно выполнить за полиномиальное время.

Чтобы доказать, что задача TSP NP-сложная, покажем, что HAM-CYCLE $\leq_{\rm P}$ TSP. Пусть $G = (V, E)$ — экземпляр задачи HAM-CYCLE. Экземпляр TSP конструируется следующим образом. Сформируем полный граф $G' = (V, E')$, где $E' = \{(i,j) : i, j \in V \text{ и } i \neq j\}$. Определим также функцию стоимости c как

$$c(i,j) = \begin{cases} 0 \text{ , } & \text{если } (i,j) \in E \text{ ,} \\ 1 \text{ , } & \text{если } (i,j) \notin E \text{ .} \end{cases}$$

(Заметим, что, поскольку граф G неориентированный, в нем отсутствуют петли, поэтому для всех вершин $v \in V$ выполняется равенство $c(v, v) = 1$.) Тогда в качестве экземпляра TSP выступает набор $\langle G', c, 0 \rangle$, который легко составить за полиномиальное время.

Покажем теперь, что граф G содержит гамильтонов цикл тогда и только тогда, когда граф G' включает в себя тур, стоимость которого не превышает 0. Предположим, что граф G содержит гамильтонов цикл h. Каждое ребро цикла h принадлежит множеству E, поэтому его стоимость в графе G' равна 0. Таким образом, стоимость цикла h в графе G' равна 0. И обратно — предположим, что граф G' содержит тур h', стоимость которого не превышает 0. Поскольку стоимость ребер графа G' равна 0 или 1, стоимость тура h' равна 0, и стоимость каждого ребра из тура должна равняться 0. Таким образом, тур h' содержит только ребра из множества E. Можно сделать вывод, что тур h' — это гамильтонов цикл в графе G. ■

34.5.5. Задача о сумме подмножества

Следующая NP-полная задача, которая будет рассмотрена, принадлежит к разряду арифметических. В ***задаче о сумме подмножества*** (subset-sum problem) задаются конечное множество S положительных целых чисел и ***целевое значение*** (target) $t > 0$. Спрашивается, существует ли подмножество $S' \subseteq S$, сумма элементов которого равна t. Например, если $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ и $t = 138457$, то решением является подмножество $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$.

Как обычно, определим задачу как язык:

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{имеется } S' \subseteq S, \text{ такое, что } t = \textstyle\sum_{s \in S'} s\} \ .$$

Как в любой арифметической задаче, важно помнить, что в нашем стандартном коде предполагается, что входные целые числа кодируются бинарными значениями. При этом предположении можно показать, что вероятность наличия быстрого алгоритма, позволяющего решить задачу о сумме подмножества, очень мала.

Теорема 34.15

Задача о сумме подмножества является NP-полной.

Доказательство. Чтобы показать, что задача SUBSET-SUM принадлежит классу NP, для экземпляра $\langle S, t\rangle$ выберем в качестве сертификата подмножество S'. Проверку равенства $t = \sum_{s \in S'} s$ в алгоритме верификации можно выполнить за полиномиальное время.

Теперь покажем, что 3-CNF-SAT $\leq_{\text{P}}$ SUBSET-SUM. Если задана 3-CNF-формула ϕ, зависящая от переменных $x_1, x_2, \ldots, x_n$ и содержащая подвыражения $C_1, C_2, \ldots, C_k$, в каждое из которых входит ровно по три различных литерала, то в алгоритме приведения строится такой экземпляр $\langle S, t\rangle$ задачи о сумме подмножества, что формула ϕ выполнима тогда и только тогда, когда существует подмножество S, сумма элементов которого равна t. Не теряя общности, можно сделать два упрощающих предположения о формуле ϕ. Во-первых, ни в одном подвыражении не содержится одновременно и переменная, и ее отрицание, потому что такое подвыражение автоматически выполняется при любых значениях этой переменной. Во-вторых, каждая переменная входит хотя бы в одно подвыражение, так как в противном случае безразлично, какое значение ей присваивается.

В процессе приведения в множестве S создается по два числа для каждой переменной x_i и для каждого выражения C_j. Эти числа будут создаваться в десятичной системе счисления, и все они будут содержать по $n + k$ цифр, каждая из которых соответствует переменной или подвыражению в скобках. Основание системы счисления 10 (и, как мы увидим, и другие основания) обладает свойством, необходимым для предотвращения переноса значений из младших разрядов в старшие.

Как видно из рис. 34.19, множество S и целевое значение t конструируются следующим образом. Присвоим каждому разряду числа метку, соответствующую либо переменной, либо подвыражению в скобках. Цифры, которые находятся в k младших разрядах, помечены подвыражениями, а цифры в n старших разрядах помечены переменными.

- Все цифры целевого значения t, помеченные переменными, равны 1, а помеченные подвыражениями — равны 4.
- Для каждой переменной x_i в множестве S содержатся два целых числа — v_i и v'_i. В каждом из них в разряде с меткой x_i находится значение 1, а в разрядах, соответствующих всем другим переменным, — значение 0. Если литерал x_i входит в подвыражение C_j, то цифра с меткой C_j в числе v_i равна 1. Если же

подвыражение C_j содержит литерал $\neg x_i$, цифра с меткой C_j в числе v'_i равна 1. Все остальные цифры, метки которых соответствуют другим подвыражениям, в числах v_i и v'_i равны 0.

Значения v_i и v'_i в множестве S не повторяются. Почему? Если $l \neq i$, то ни v_l, ни v'_l не могут быть равны v_i или v'_i в n старших разрядах. Кроме того, согласно сделанным выше упрощающим предположениям, никакие v_i и v'_i не могут быть равны во всех k младших разрядах. Если бы числа v_i и v'_i были равны, то литералы x_i и $\neg x_i$ должны были бы входить в один и тот же набор подвыражений в скобках. Однако согласно предположению ни одно подвыражение не содержит одновременно и x_i, и $\neg x_i$, и хотя бы один из этих литералов входит в одно из подвыражений. Поэтому должно существовать какое-то подвыражение C_j, для которого v_i и v'_i различаются.

- Для каждого подвыражения C_j в множестве S содержатся два целых числа s_j и s'_j. Каждое из них содержит нули во всех разрядах, метки которых отличаются от C_j. В числе s_j цифра с меткой C_j равна 1, а в числе s'_j эта цифра равна 2. Такие целые числа выступают в роли "фиктивных переменных", которые мы используем для того, чтобы получить в каждом разряде, помеченном подвыражением, целевое значение 4.

  Беглый взгляд на пример, проиллюстрированный на рис. 34.19, показывает, что никакие значения s_j и s'_j в множестве S не повторяются.

Заметим, что максимальная сумма цифр в каждом из разрядов равна 6, что и осуществляется в цифрах, метки которых соответствуют подвыражениям (три единицы от значений v_i и v'_i плюс 1 и 2 от значений s_j и s'_j). Потому эти значения интерпретируются как числа в десятичной системе счисления, — чтобы не было переноса из младших разрядов в старшие[11].

Такое приведение можно выполнить за полиномиальное время. Множество S содержит $2n + 2k$ значений, в каждом из которых по $n + k$ цифр, поэтому время, необходимое для получения всех цифр, выражается полиномиальной функцией от $n + k$. Целевое значение t содержит $n + k$ цифр, и в процессе приведения каждую из них можно получить за фиксированное время.

Теперь покажем, что 3-CNF-формула ϕ выполнима тогда и только тогда, когда существует подмножество $S' \subseteq S$, сумма элементов которого равна t. Сначала предположим, что у формулы ϕ имеется выполняющий набор. Если в нем $x_i = 1$ (где $i = 1, 2, \ldots, n$), то число v_i включается в множество S'. В противном случае в это множество включается число v'_i. Другими словами, в множество S' включаются именно те значения v_i и v'_i, которым в выполняющем наборе соответствуют литералы, равные единице. Включая в множество S' либо число v_i, либо число v'_i, но не оба этих числа для всех i, и помещая в значениях s_j и s'_j нули во все разряды с метками, соответствующими переменным, мы видим, что сумма цифр

---

[11] Фактически подошло бы любое основание $b \geq 7$. Так, экземпляр задачи, который рассматривается в начале этого подраздела, представляет собой множество S и целевое значение t на рис. 34.19, где все значения рассматриваются как числа в семеричной системе счисления, а множество S перечислено в порядке убывания.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
[illegible]	[illegible]	0	[illegible]	0	0	[illegible]	0	[illegible]
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
[illegible]	[illegible]	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
[illegible]	[illegible]	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
[illegible]	[illegible]	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_1	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Рис. 34.19. Приведение 3-CNF-SAT к SUBSET-SUM. 3-CNF-формула имеет вид $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, где $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ и $C_4 = (x_1 \vee x_2 \vee x_3)$. Выполняющий набор ϕ представляет собой $\langle x_1 = 0,\ x_2 = 0,\ x_3 = 1\rangle$. Множество S, полученное в процессе приведения, состоит из представленных в таблице чисел в десятичной системе счисления; считывая их сверху вниз, находим, что $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. Целевое значение t равно 1114444. Подмножество $S' \subseteq S$ выделено светлым цветом и содержит v'_1, v'_2 и v_3, соответствующие выполняющему набору. В него также входят фиктивные переменные s_1, s'_1, s'_2, s_3, s_4 и s'_4, обеспечивающие получение значений 4 в цифрах целевого значения, помеченных подвыражениями C_1–C_4.

в этих разрядах по всем значениям множества S' должна быть равной единице, что совпадает с цифрами в этих разрядах целевого значения t. Поскольку все подвыражения в скобках выполняются, в каждом таком подвыражении имеется литерал, значение которого равно единице. Поэтому каждая цифра, помеченная подвыражением, включает как минимум одну единицу, вносимую в сумму благодаря вкладу значения v_i или значения v'_i из множества S'. В действительности в каждом подвыражении единице могут быть равны 1, 2 или 3 литерала, поэтому в каждом разряде, соответствующем подвыражению, сумма цифр по значениям v_i и v'_i множества S' равна 1, 2 или 3. Так, в примере, проиллюстрированном на рис. 34.19, в выполняющем наборе единице равны литералы $\neg x_1$, $\neg x_2$ и x_3. Каждое из подвыражений C_1 и C_4 содержит ровно по одному из этих литералов, поэтому числа v'_1, v'_2 и v_3 в совокупности дают единичный вклад в цифры, соответствующие подвыражениям C_1 и C_4. Подвыражение C_2 содержит два из этих литералов, поэтому числа v'_1, v'_2 и v_3 в совокупности дают вклад 2 в цифру, соответствующую C_2. Подвыражение C_3 содержит все три перечисленных литерала, и числа v'_1, v'_2 и v_3 дают вклад 3 в цифру, помеченную подвыражением C_3. Целевое значение, равное 4, достигается в каждом разряде с меткой, отвечающей подвыражению C_j, путем добавления в множество S' непустого множества

из соответствующих фиктивных переменных $\{s_j, s'_j\}$. На рис. 34.19 множество S' включает значения s_1, s'_1, s'_2, s_3, s_4 и s'_4. Поскольку сумма цифр по всем значениям множества S' во всех разрядах совпадает с соответствующими цифрами целевого значения t, а при суммировании не производился перенос значений из младших разрядов в старшие, то сумма значений множества S' равна t.

Теперь предположим, что имеется подмножество $S' \subseteq S$, сумма элементов которого равна t. Для каждого значения $i = 1, 2, \ldots, n$ это подмножество должно включать в себя ровно по одному из значений v_i или v'_i, потому что в противном случае сумма цифр в разрядах, соответствующих переменным, не была бы равна единице. Если $v_i \in S'$, то выполняем присваивание $x_i = 1$. В противном случае $v'_i \in S'$, и выполняется присваивание $x_i = 0$. Мы утверждаем, что в результате такого присваивания выполняется каждое подвыражение C_j, $j = 1, 2, \ldots, k$. Чтобы доказать это утверждение, заметим, что для того, чтобы сумма цифр в разряде с меткой C_j была равна 4, подмножество S' должно включать хотя бы одно из значений v_i или v'_i, в котором единица находится в разряде с меткой C_j, так как суммарный вклад фиктивных переменных s_j и s'_j не превышает 3. Если подмножество S' включает в себя значение v_i, в котором в разряде с меткой C_j содержится единица, то в подвыражение C_j входит литерал x_i. Поскольку при $v_i \in S'$ выполняется присваивание $x_i = 1$, подвыражение C_j выполняется. Если множество S' включает в себя значение v'_i, в котором в этом разряде содержится единица, то в подвыражение C_j входит литерал $\neg x_i$. Так как при $v'_i \in S'$ выполняется присваивание $x_i = 0$, подвыражение C_j снова выполняется. Таким образом, в формуле ϕ выполняются все подвыражения, и на этом доказательство теоремы завершается. ∎

Упражнения

34.5.1

В ***задаче об изоморфизме подграфу*** (subgraph-isomorphism problem) задаются два графа (G_1 и G_2) и спрашивается, изоморфен ли граф G_1 какому-либо подграфу графа G_2. Покажите, что эта задача NP-полная.

34.5.2

В ***задаче 0-1 целочисленного программирования*** (0-1 integer-programming problem) задаются целочисленная матрица A размером $m \times n$ и целочисленный m-компонентный вектор b и спрашивается, существует ли целочисленный n-компонентный вектор x, элементы которого являются элементами множества $\{0, 1\}$, такой, что $Ax \leq b$. Докажите, что задача 0-1 целочисленного программирования является NP-полной. (*Указание:* приведите к этой задаче задачу 3-CNF-SAT.)

34.5.3

Задача целочисленного линейного программирования (integer linear-programming problem) похожа на задачу 0-1 целочисленного программирования, описанную в упр. 34.5.2, но в ней компоненты вектора x могут быть любыми целыми числами, а не только нулем и единицей. Исходя из предположения, согласно которому

задача 0-1 целочисленного программирования является NP-полной, покажите, что задача целочисленного линейного программирования также NP-полная.

34.5.4

Покажите, как решить задачу о сумме подмножества за полиномиальное время, если целевое значение t выражено в унарной системе счисления, т.е. представлено последовательностью из t единиц.

34.5.5

В ***задаче о разбиении множества*** (set-partition problem) в качестве входных данных выступает множество чисел S. Спрашивается, можно ли это числа разбить на два множества (A и $\overline{A} = S - A$) таким образом, чтобы $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Покажите, что задача о разбиении множества является NP-полной.

34.5.6

Покажите, что задача о гамильтоновом пути NP-полная.

34.5.7

Задача о самом длинном простом цикле (longest-simple-cycle problem) — это задача, в которой в заданном графе выполняется поиск простого (без повторения вершин) цикла максимальной длины. Покажите, что эта задача — NP-полная.

34.5.8

В ***половинной задаче о 3-CNF-выполнимости*** (half 3-CNF satisfiability problem) задается 3-CNF-формула ϕ с n переменными и m подвыражениями, где m — четное. Нужно определить, существует ли набор значений переменных формулы ϕ, при котором результат ровно половины подвыражений в скобках равен нулю, а результат второй половины этих подвыражений равен единице. Докажите, что половинная задача о 3-CNF-выполнимости является NP-полной.

Задачи

34.1. Независимое множество

Независимым множеством (independent set) графа $G = (V, E)$ называется такое подмножество вершин $V' \subseteq V$, что каждое ребро из множества E инцидентно хотя бы одной вершине из множества V'. ***Задача о независимом множестве*** (independent-set problem) заключается в том, чтобы найти в заданном графе G независимое множество максимального размера.

а. Сформулируйте задачу принятия решения, соответствующую задаче о независимом множестве, и докажите, что она является NP-полной. (*Указание:* приведите к этой задаче задачу о клике.)

б. Предположим, что для задачи принятия решения, определенной в п. (а), имеется подпрограмма в виде “черного ящика”, решающего эту задачу. Сформулируйте алгоритм поиска независимого множества, имеющего максимальный

размер. Время работы этого алгоритма должно выражаться полиномиальной функцией от величин $|V|$ и $|E|$. При этом предполагается, что каждый запрос к черному ящику учитывается как одна операция.

Несмотря на то что задача о независимом множестве является NP-полной, некоторые частные ее случаи разрешимы за полиномиальное время.

в. Разработайте эффективный алгоритм, позволяющий решить задачу о независимом множестве, если степень каждой вершины графа G равна 2. Проанализируйте время работы этого алгоритма и докажите его корректность.

г. Разработайте эффективный алгоритм, позволяющий решить задачу о независимом множестве для двудольного графа G. Проанализируйте время работы этого алгоритма и докажите его корректность. (*Указание:* воспользуйтесь результатами, полученными в разделе 26.3.)

34.2. Бонни и Клайд

Бонни и Клайд только что ограбили банк. У них есть мешок денег, который нужно разделить. В каждом из описанных ниже сценариев требуется либо сформулировать алгоритм с полиномиальным временем работы, либо доказать, что задача NP-полная. В каждом случае в качестве входных данных выступает список, состоящий из n содержащихся в мешке элементов, а также стоимость каждого из них.

а. В мешке n монет двух различных номинаций: одни монеты стоят x долларов, а другие — y долларов. Деньги следует разделить поровну.

б. В мешке n монет произвольного количества различных номиналов, но при этом каждый номинал является неотрицательной целой степенью двойки, т.е. возможны такие номиналы: 1 доллар, 2 доллара, 4 доллара и т.д. Деньги следует разделить поровну.

в. В мешке лежат n чеков, по невероятному совпадению выписанных на имя "Бонни или Клайд". Нужно разделить чеки таким образом, чтобы по ним можно было получить одинаковые суммы.

г. Как и в случае (в), в мешке лежит n чеков, но на этот раз допускается расхождение в суммах, которое не должно превышать ста долларов.

34.3. Раскраска графов

Художник, изготавливающий карту, пытается использовать минимально возможное количество красок для раскраски стран на карте так, чтобы никакие смежные страны не были окрашены в один и тот же цвет. Эту задачу можно моделировать с помощью неориентированного графа $G = (V, E)$, в котором каждая вершина представляет страну, при этом страны, имеющие с ней общую границу, представлены вершинами, смежными с ней. Тогда ***k*-раскрашивание** представляет собой функцию $c : V \to \{1, 2, \dots, k\}$, такую, что $c(u) \neq c(v)$ для каждого ребра

37 Зак. 3726

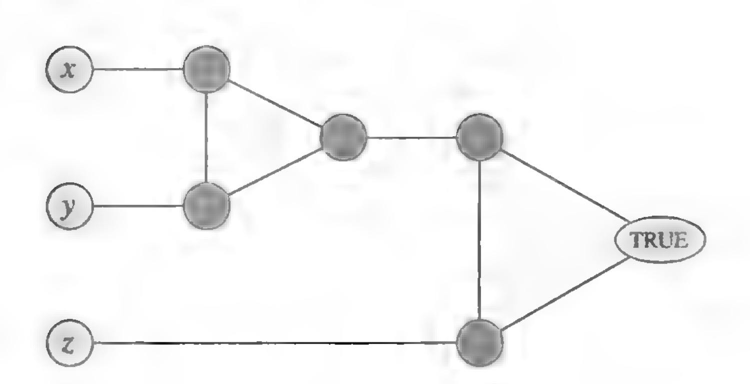


Рис. 34.20. Структурный элемент, который соответствует подвыражению $(x \vee y \vee z)$ и используется в задаче 34.3.

$(u, v) \in E$. Другими словами, числа $1, 2, \ldots, k$ представляют k цветов, и смежные вершины должны иметь разные цвета. ***Задача о раскраске графа*** (graph-coloring problem) состоит в определении минимального количества цветов, необходимых для раскраски данного графа.

а. Разработайте эффективный алгоритм 2-раскрашивания графа, если такое раскрашивание возможно.

б. Переформулируйте задачу о раскрашивании графов в виде задачи принятия решения. Покажите, что эта задача разрешима за полиномиальное время тогда и только тогда, когда за полиномиальное время разрешима задача о раскрашивании графов.

в. Пусть язык 3-COLOR представляет собой множество графов, для которых возможно 3-раскрашивание. Покажите, что если задача 3-COLOR — NP-полная, то задача принятия решения из п. (б) также NP-полная.

Чтобы доказать NP-полноту задачи 3-COLOR, воспользуемся приведением к этой задаче задачи 3-CNF-SAT. Для заданной формулы ϕ, содержащей m подвыражений в скобках и n переменных $x_1, x_2, \ldots, x_n$, конструируется граф $G = (V, E)$ описанным ниже способом. Множество V содержит по одной вершине для каждой переменной, по одной вершине для каждого отрицания переменной, по 5 вершин для каждого подвыражения и 3 специальные вершины: TRUE, FALSE и RED. Ребра в этом графе могут быть двух типов: "литеральные" ребра, которые не зависят от подвыражений в скобках, и "дизъюнктивные" ребра, которые от них зависят. Литеральные ребра образуют треугольник на специальных вершинах, а также треугольник на вершинах x_i, $\neg x_i$ и RED для $i = 1, 2, \ldots, n$.

г. Докажите, что при любом 3-раскрашивании c графа, состоящего из литеральных ребер, из каждой пары вершин x_i, $\neg x_i$ одна окрашена как $c(\text{TRUE})$, а другая — как $c(\text{FALSE})$. Докажите, что для любого набора значений функции ϕ существует 3-раскрашивание графа, содержащего только литеральные ребра.

Структурный элемент, изображенный на рис. 34.20, помогает обеспечить выполнение условия, соответствующего подвыражению $(x \vee y \vee z)$. Каждое подвыражение требует своей копии пяти вершин, выделенных на рисунке темным цветом; они соединяются с литералами подвыражения и специальной вершиной TRUE.

д. Докажите, что если каждая из вершин x, y и z окрашена в один из двух цветов — $c(\text{TRUE})$ или $c(\text{FALSE})$, — то для изображенного на рисунке структурного элемента правильное 3-раскрашивание возможно тогда и только тогда, когда цвет хотя бы одной из вершин x, y и z — $c(\text{TRUE})$.

е. Завершите доказательство NP-полноты 3-COLOR.

34.4. Расписание с прибылью и предельными сроками

Предположим, что имеется одна вычислительная машина и n заданий $a_1, a_2, \ldots, a_n$. Каждое задание a_j характеризуется временем выполнения t_j (временем работы машины, необходимым для выполнения задания), прибылью p_j и конечным сроком выполнения d_j. В каждый момент времени машина может обрабатывать только одно задание, причем задание a_j после запуска должно без прерываний выполняться в течение времени t_j. Если задание a_j будет выполнено до наступления конечного срока его выполнения d_j, то будет получена прибыль p_j, но если конечный срок выполнения будет сорван, то и прибыли не будет. Сформулируем такую задачу оптимизации: пусть для множества n заданий известны время обработки, прибыль и время выполнения; требуется составить расписание таким образом, чтобы были выполнены все задания и общая сумма прибыли была максимальной. Все величины — время выполнения, прибыль и конечный срок — неотрицательные числа.

а. Сформулируйте эту задачу в виде задачи принятия решения.

б. Покажите, что эта задача принятия решения NP-полная.

в. Разработайте алгоритм с полиномиальным временем работы, позволяющий решить задачу принятия решения в предположении, что все времена выполнения выражаются целыми числами от 1 до n. (*Указание:* воспользуйтесь методами динамического программирования.)

г. Разработайте алгоритм с полиномиальным временем работы, позволяющий решить задачу оптимизации в предположении, что все времена выполнения выражаются целыми числами от 1 до n.

Заключительные замечания

Книга Гарея (Garey) и Джонсона (Johnson) [128] является замечательным учебником по вопросам NP-полноты; в ней не только подробно обсуждается теория, но и описываются многие задачи, NP-полнота которых была доказана до 1979 года. Из этой книги заимствовано доказательство теоремы 34.13, а список NP-полных задач, описанных в начале раздела 34.5, взят из содержания этой книги. В 1981–1992 годах Джонсон написал в *Journal of Algorithms* серию из 23 статей о новых достижениях в исследованиях NP-полноты. В книгах Хопкрофта (Hopcroft), Мотвани (Motwani) и Ульмана (Ullman) [176], Лью-

иса (Lewis) и Пападимитриу (Papadimitriou) [235], Пападимитриу [268] и Сипсера (Sipser) [315] проблема NP-полноты удачно трактуется в контексте теории сложности. В книгах Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [5], Дасгупты (Dasgupta), Пападимитриу (Papadimitriou) и Вазирани (Vazirani) [81], Джонсонбафа (Johnsonbaugh) и Шефера (Schaefer) [192] и Кляйнберга (Kleinberg) и Тардоса (Tardos) [207] также описывается NP-полнота и рассматриваются примеры приведений.

Класс P был введен в 1964 году Кобхемом (Cobham) [71] и независимо в 1965 году Эдмондсом (Edmonds) [99], который ввел также класс NP и выдвинул гипотезу о том, что $P \neq NP$. Понятие NP-полноты впервые было предложено Куком (Cook) [74] в 1971 году. Кук представил первое доказательство NP-полноты задач о выполнимости формул и о 3-CNF-выполнимости. Левин (Levin) [233] независимо пришел к этому понятию; он доказал NP-полноту задачи о мозаике. Карп (Karp) [198] в 1972 году предложил методику приведения и продемонстрировал богатое разнообразие NP-полных задач. В статье Карпа впервые доказывается NP-полнота задач о клике, о вершинном покрытии и о гамильтоновом цикле. С тех пор многими исследователями была доказана NP-полнота тысяч задач. В 1995 году на заседании, посвященном 60-летию Карпа, Пападимитриу в своем докладе заметил: "...каждый год выходит около шести тысяч статей, в заголовке, аннотации или списке ключевых слов которых содержится термин 'NP-полный'. Он встречается чаще, чем термины 'компилятор', 'база данных', 'экспертная система', 'нейронная сеть' или 'операционная система'".

Недавняя работа по теории сложности пролила свет на вопрос о сложности приближенных компьютерных вычислений. В ней приводится новое определение класса NP с помощью "вероятностно проверяемых доказательств". В этом определении подразумевается, что для таких задач, как задача о клике, о вершинном покрытии, о коммивояжере, в которой выполняется неравенство треугольника, и во многих других получение хорошего приближенного решения является NP-сложным, поэтому оно не легче, чем получение оптимальных решений. Введение в эту область можно найти в диссертации Ароры (Arora) [20], в главе Ароры и Лунда (Lund) в [171], в обзорной статье Ароры [21], в книге под редакцией Мэйра (Mayr), Прёмеля (Prömel) и Стиджера (Steger) [244], а также в обзорной статье Джонсона (Johnson) [190].

Глава 35. Приближенные алгоритмы

Многие задачи, представляющие практический интерес, являются NP-полными. Однако они слишком важны, чтобы отказаться от их решения лишь на том основании, что неизвестно, как получить их оптимальное решение за полиномиальное время. Но даже для NP-полных задач остается надежда. Во-первых, если объем входных данных небольшой, алгоритм, время работы которого выражается показательной функцией, вполне может подойти. Во-вторых, иногда удается выделить важные частные случаи, разрешимые за полиномиальное время. В-третьих, остается возможность найти за полиномиальное время решение, *близкое к оптимальному* (в наихудшем либо в среднем случае). На практике такие решения часто являются достаточно хорошими. Алгоритм, возвращающий решения, близкие к оптимальным, называется ***приближенным алгоритмом*** (approximation algorithm). В этой главе описаны приближенные алгоритмы с полиномиальным временем выполнения, предназначенные для решения некоторых NP-полных задач.

Оценка качества приближенных алгоритмов

Предположим, мы работаем над задачей оптимизации, каждому из возможных решений которой сопоставляется положительная стоимость, и требуется найти решение, близкое к оптимальному. В зависимости от задачи оптимальное решение можно определить либо как такое, которому соответствует максимально возможная стоимость, либо как такое, которому соответствует минимально возможная стоимость; другими словами, это может быть либо задача максимизации, либо задача минимизации.

Говорят, что алгоритм решения задачи обладает ***отношением, или коэффициентом, аппроксимации (приближения)*** (approximation ratio) $\rho(n)$, если для произвольных входных данных размером n стоимость C решения, полученного в результате выполнения этого алгоритма, отличается от стоимости C^* оптимального решения не более чем в $\rho(n)$ раз:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) . \qquad (35.1)$$

Алгоритм, в котором достигается коэффициент аппроксимации $\rho(n)$, будем называть ***$\rho(n)$-приближенным алгоритмом*** ($\rho(n)$-approximation algorithm). Опре-

деления коэффициента аппроксимации и $\rho(n)$-приближенного алгоритма применимы и к задачам минимизации, и к задачам максимизации. Для задач максимизации выполняется неравенство $0 < C \leq C^*$, и отношение C^*/C равно величине, на которую стоимость оптимального решения больше стоимости приближенного решения. Аналогично для задач минимизации выполняется неравенство $0 < C^* \leq C$, и отношение C/C^* дает ответ, во сколько раз стоимость приближенного решения больше стоимости оптимального решения. Поскольку предполагается, что стоимости всех решений положительные, эти коэффициенты вполне определены. Коэффициент аппроксимации приближенного алгоритма не может быть меньше 1, поскольку из неравенства $C/C^* < 1$ следует неравенство $C^*/C > 1$. Таким образом, 1-приближенный алгоритм[1] выдает оптимальное решение, а приближенный алгоритм с бóльшим отношением аппроксимации может возвратить решение, которое намного хуже оптимального.

Для многих задач разработаны приближенные алгоритмы с полиномиальным временем работы и малыми постоянными отношениями аппроксимации. Есть также задачи, для которых лучшие из известных приближенных алгоритмов с полиномиальным временем работы характеризуются коэффициентами аппроксимации, величина которых возрастает с ростом размера входных данных n. Примером такой задачи является задача о покрытии множества, представленная в разделе 35.3.

Некоторые NP-полные задачи допускают наличие приближенных алгоритмов с полиномиальным временем работы, коэффициент аппроксимации которых можно уменьшать за счет увеличения времени их работы. Другими словами, в них допускается компромисс между временем вычисления и качеством приближения. В качестве примера можно привести задачу о сумме подмножества, которая исследуется в разделе 35.5. Эта ситуация достаточно важна и заслуживает собственного имени.

Схема аппроксимации (approximation scheme) задачи оптимизации — это приближенный алгоритм, входные данные которого включают в себя не только параметры экземпляра задачи, но и такое значение $\epsilon > 0$, что для любого фиксированного значения ϵ эта схема является $(1 + \epsilon)$-приближенным алгоритмом. Схему аппроксимации называют ***схемой аппроксимации с полиномиальным временем выполнения*** (polynomial-time approximation scheme), если для любого фиксированного значения $\epsilon > 0$ работа этой схемы завершается за время, выраженное полиномиальной функцией от размера n входных данных.

Время работы схемы аппроксимации с полиномиальным временем вычисления может очень быстро возрастать при уменьшении величины ϵ. Например, это время может вести себя как $O(n^{2/\epsilon})$. В идеале, если величина ϵ уменьшается на постоянный множитель, время, необходимое для достижения нужного приближения, не должно возрастать более чем на постоянный множитель (хотя и не обязательно на тот же, на который уменьшается значение ϵ).

---

[1]Если коэффициент аппроксимации не зависит от n, мы используем термины "отношение аппроксимации ρ" и "ρ-приближенный алгоритм", указывающие на отсутствие зависимости от n.

Говорят, что схема аппроксимации является ***схемой аппроксимации с полностью полиномиальным временем работы*** (fully polynomial-time approximation scheme), если время ее работы выражается полиномом как от $1/\epsilon$, так и от размера входных данных задачи n. Например, время работы такой схемы может вести себя как $O((1/\epsilon)^2 n^3)$. В такой схеме любое уменьшение величины ϵ на постоянный множитель сопровождается соответствующим увеличением времени работы на постоянный множитель.

Краткое содержание главы

В первых четырех разделах этой главы приведены некоторые примеры приближенных алгоритмов с полиномиальным временем работы, позволяющие получать приближенные решения NP-полных задач. В пятом разделе представлена схема аппроксимации с полностью полиномиальным временем работы. Начало раздела 35.1 посвящено исследованию задачи о вершинном покрытии, которая относится к классу NP-полных задач минимизации. Для этой задачи существует приближенный алгоритм, характеризующийся коэффициентом аппроксимации 2. В разделе 35.2 представлен приближенный алгоритм с коэффициентом аппроксимации 2, предназначенный для решения частного случая задачи коммивояжера, когда функция стоимости удовлетворяет неравенству треугольника. Также показано, что если неравенство треугольника не соблюдается, то для любой константы $\rho \geq 1$ существование ρ-приближенного алгоритма связано с выполнением условия P = NP. В разделе 35.3 показано, как использовать жадный метод в качестве эффективного приближенного алгоритма для решения задачи о покрытии множества. При этом возвращается покрытие, стоимость которого в наихудшем случае превышает оптимальную на множитель, выражающийся логарифмической функцией. В разделе 35.4 представлены еще два приближенных алгоритма. В первом из них исследуется оптимизирующая версия задачи о 3-CNF-выполнимости и приводится простой рандомизированный алгоритм, который выдает решение, характеризующееся ожидаемым коэффициентом аппроксимации, равным $8/7$. Затем изучается взвешенный вариант задачи о вершинном покрытии и описывается, как с помощью методов линейного программирования разработать 2-приближенный алгоритм. Наконец в разделе 35.5 представлена схема аппроксимации с полностью полиномиальным временем работы, предназначенная для решения задачи о сумме подмножества.

35.1. Задача о вершинном покрытии

Задача о вершинном покрытии определена в разделе 34.5.2. В этом же разделе доказано, что эта задача является NP-полной. Напомним, что ***вершинное покрытие*** (vertex cover) неориентированного графа $G = (V, E)$ — это такое подмножество $V' \subseteq V$, что если (u, v) — ребро графа G, то либо $u \in V'$, либо $v \in V'$, либо

справедливы оба эти соотношения. Размером вершинного покрытия называется количество содержащихся в нем вершин.

Задача о вершинном покрытии (vertex-cover problem) состоит в том, чтобы найти для заданного неориентированного графа вершинное покрытие минимального размера. Назовем такое вершинное покрытие ***оптимальным вершинным покрытием*** (optimal vertex cover). Эта задача представляет собой оптимизирующую версию NP-полной задачи принятия решения.

Несмотря на то что мы не знаем, как найти оптимальное вершинное покрытие графа G за полиномиальное время, не так сложно найти вершинное покрытие, близкое к оптимальному. Приведенный ниже приближенный алгоритм принимает в качестве входных данных параметры неориентированного графа G и возвращает вершинное покрытие, размер которого превышает размер оптимального вершинного покрытия не более чем в два раза.

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      Пусть (u, v) — произвольное ребро из E'
5      C = C ∪ {u, v}
6      Удалить из E' все ребра, инцидентные u или v
7  return C
```

На рис. 35.1 показано, как алгоритм APPROX-VERTEX-COVER строит вершинное покрытие для демонстрационного графа. Переменная C содержит создаваемое вершинное покрытие. В строке 1 выполняется инициализация C пустым множеством. В строке 2 в множество E' копируется множество ребер $G.E$ графа. Цикл в строках 3–6 поочередно выбирает ребра (u, v) из множества E', добавляет их конечные точки u и v в C и удаляет из E' все ребра, покрываемые либо u, либо v. Наконец в строке 7 возвращается вершинное покрытие C. Время работы этого алгоритма равно $O(V + E)$ при использовании для представления E' списков смежности.

Теорема 35.1

Алгоритм APPROX-VERTEX-COVER является 2-приближенным алгоритмом с полиномиальным временем работы.

Доказательство. Мы уже показали, что алгоритм APPROX-VERTEX-COVER имеет полиномиальное время работы.

Множество вершин C, которое возвращается алгоритмом APPROX-VERTEX-COVER, является вершинным покрытием, поскольку алгоритм не выходит из цикла, пока каждое ребро $G.E$ не будет покрыто некоторой вершиной из множества C.

Чтобы показать, что рассматриваемый алгоритм возвращает вершинное покрытие, размер которого превышает размер оптимального вершинного покрытия не более чем в два раза, обозначим через A множество ребер, выбираемых в стро-

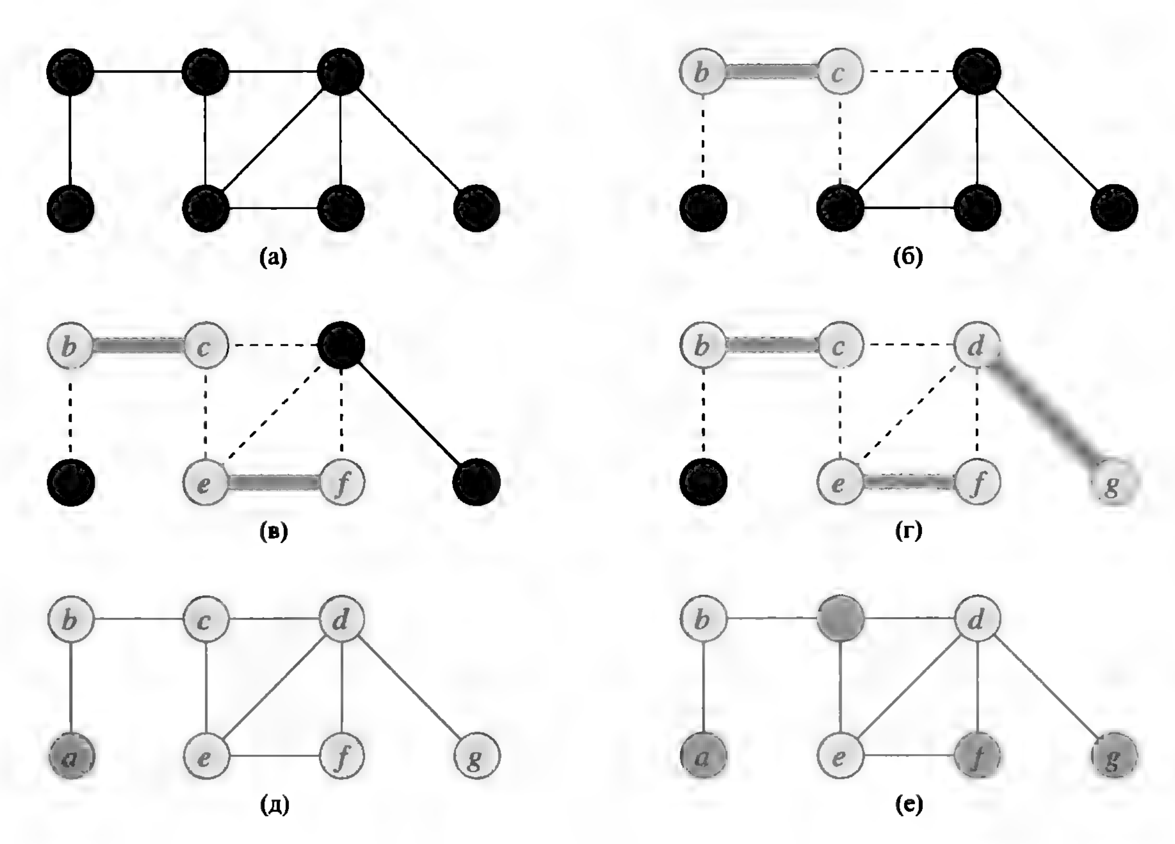


Рис. 35.1. Работа алгоритма APPROX-VERTEX-COVER. **(а)** Исходный граф G с семью вершинами и восемью ребрами. **(б)** Ребро (b, c), выделенное серым цветом, — первое по счету ребро, выбранное алгоритмом APPROX-VERTEX-COVER. Вершины b и c, показанные светло-серой штриховкой, добавляются в множество C, в котором содержится создаваемое вершинное покрытие. Показанные пунктиром ребра (a, b), (c, e) и (c, d) удаляются, поскольку они уже покрыты вершинами из множества C. **(в)** Алгоритмом выбрано ребро (e, f), а вершины e и f добавлены в множество C. **(г)** Алгоритмом выбрано ребро (d, g), а вершины d и g добавлены в множество C. **(д)** Множество C, которое представляет собой вершинное покрытие, полученное в результате выполнения алгоритма APPROX-VERTEX-COVER. Оно состоит из шести вершин — b, c, d, e, f, g. **(е)** Оптимальное вершинное покрытие для рассмотренного экземпляра задачи, состоящее всего из трех вершин: b, d и e.

ке 4 алгоритма APPROX-VERTEX-COVER. Чтобы покрыть ребра множества A, каждое вершинное покрытие, в том числе и оптимальное покрытие C^*, должно содержать хотя бы одну конечную точку каждого ребра из множества A. Никакие два ребра из этого множества не имеют общих конечных точек, поскольку после того, как ребро выбирается в строке 4, все другие ребра с такими же конечными точками удаляются из множества E' в строке 6. Таким образом, никакие два ребра из множества A не покрываются одной и той же вершиной из множества C^*, из чего следует, что нижняя граница размера оптимального вершинного покрытия равна

$$|C^*| \geq |A| . \qquad (35.2)$$

При каждом выполнении строки 4 выбирается ребро, ни одна из конечных точек которого пока еще не вошла в множество C. Это позволяет оценить сверху (фактически указать точную верхнюю границу) размер возвращаемого вершинного покрытия:

$$|C| = 2|A| . \qquad (35.3)$$

Сопоставив уравнения (35.2) и (35.3), получаем

$$\begin{aligned} |C| &= 2\,|A| \\ &\le 2\,|C^*| \ , \end{aligned}$$

что и доказывает теорему. ■

Еще раз вернемся к приведенному выше доказательству. На первый взгляд, может показаться удивительным, как можно доказать, что размер вершинного покрытия, возвращенного процедурой APPROX-VERTEX-COVER, не более чем в два раза превышает размер оптимального вершинного покрытия, если неизвестно, чему равен размер оптимального вершинного покрытия. Это становится возможным благодаря использованию нижней границы оптимального вершинного покрытия. Как предлагается показать в упр. 35.1.2, множество A, состоящее из ребер, выбранных в строке 4 процедуры APPROX-VERTEX-COVER, фактически является максимальным паросочетанием вершин в графе G. (***Максимальное паросочетание*** (maximal matching) — это паросочетание, которое не является собственным подмножеством ни одного другого паросочетания.) Как было показано при доказательстве теоремы 35.1, размер максимального паросочетания равен нижней границе размера оптимального вершинного покрытия. Алгоритм возвращает вершинное покрытие, размер которого не более чем в два раза превышает размер максимального паросочетания множества A. Составив отношение размера возвращаемого решения к полученной нижней границе, находим коэффициент аппроксимации. Эта методика будет использоваться и в следующих разделах.

Упражнения

35.1.1

Приведите пример графа, для которого процедура APPROX-VERTEX-COVER всегда возвращает неоптимальное решение.

35.1.2

Докажите, что множество ребер, выбранных в строке 4 процедуры APPROX-VERTEX-COVER образует максимальное паросочетание в графе G.

35.1.3 ★

Профессор предложил такую эвристическую схему решения задачи о вершинном покрытии. Одна за другой выбираются вершины с максимальной степенью, и для каждой из них удаляются все инцидентные ребра. Приведите пример, демонстрирующий, что коэффициент аппроксимации, предложенной профессором, превышает 2. (*Указание:* попытайтесь построить двудольный граф, вершины в левой части которого имеют одинаковые степени, а вершины в правой части — разные степени.)

35.1.4

Разработайте эффективный жадный алгоритм, позволяющий найти оптимальное вершинное покрытие дерева за линейное время.

35.1.5

Из доказательства теоремы 34.12 известно, что задача о вершинном покрытии и NP-полная задача о клике являются взаимодополняющими в том смысле, что оптимальное вершинное покрытие — это дополнение к клике максимального размера в дополняющем графе. Следует ли из этого, что для задачи о клике существует приближенный алгоритм с полиномиальным временем решения, обладающий постоянным коэффициентом аппроксимации? Обоснуйте свой ответ.

35.2. Задача о коммивояжере

Обратимся к задаче о коммивояжере, представленной в разделе 34.5.4. В ней задается полный неориентированный граф $G = (V, E)$, каждому из ребер $(u, v) \in E$ которого сопоставляется неотрицательная целочисленная стоимость $c(u, v)$, и в графе G требуется найти гамильтонов цикл минимальной стоимости. Введем дополнительное обозначение $c(A)$ — полную стоимость всех ребер подмножества $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v) .$$

Во многих практических ситуациях наиболее дешевый переход из места u в место w — по прямой, так что, если по пути зайти в какой-нибудь промежуточный путь v, это не может привести к уменьшению стоимости. Выражаясь другими словами, если срезать путь, пропустив какой-нибудь промежуточный пункт, это никогда не станет причиной увеличения стоимости. Формализуем это понятие, выдвинув утверждение, что функция стоимости c удовлетворяет ***неравенству треугольника*** (triangle inequality), если для всех вершин $u, v, w \in V$

$$c(u, w) \leq c(u, v) + c(v, w) .$$

Неравенство треугольника имеет естественный характер и автоматически удовлетворяется во многих приложениях. Например, если вершины графа — точки на плоскости, а стоимость перехода от одной вершины к другой выражается обычным евклидовым расстоянием между ними, то неравенство треугольника выполняется. (Кроме евклидова расстояния, существует множество других функций стоимости, удовлетворяющих неравенству треугольника.)

Как видно из упр. 35.2.2, задача о коммивояжере является NP-полной даже в том случае, если потребовать, чтобы функция стоимости удовлетворяла неравенству треугольника. Таким образом, мало надежд найти алгоритм с полиномиальным временем работы, позволяющий получить точное решение этой задачи. Поэтому есть смысл заняться поиском хороших приближенных алгоритмов.

В разделе 35.2.1 исследуется 2-приближенный алгоритм, позволяющий решить задачу о коммивояжере, в которой выполняется неравенство треугольника. В разделе 35.2.2 будет показано, что без соблюдения неравенства треугольника приближенный алгоритм с полиномиальным временем работы, характеризующийся постоянным коэффициентом аппроксимации, не существует, если только не окажется справедливым соотношение P = NP.

35.2.1. Задача о коммивояжере с неравенством треугольника

Применив методику из предыдущего раздела, сначала вычислим структуру — минимальное остовное дерево, — вес которой является нижней границей длины оптимального тура коммивояжера. Затем с помощью этого минимального остовного дерева создадим тур, стоимость которого не более чем в два раза превышает вес этого дерева при условии, что функция стоимости удовлетворяет неравенству треугольника. Этот подход реализован в приведенном ниже алгоритме, в котором используется алгоритм построения минимального остовного дерева MST-PRIM, описанный в разделе 23.2. Параметр G представляет собой полный неориентированный граф, а функция стоимости удовлетворяет неравенству треугольника.

APPROX-TSP-TOUR(G, c)

1 Выбираем вершину $r \in G.V$ в качестве "корневой"
2 Вычисляем минимальное остовное дерево T для G из корня r
  с использованием алгоритма MST-PRIM(G, c, r)
3 Пусть H — список вершин, упорядоченный в соответствии с
  моментом первого посещения при обходе дерева T
  в прямом порядке
4 **return** гамильтонов цикл H

Вспомним из раздела 12.1, что при прямом обходе дерева рекурсивно посещаются все его вершины, причем вершина заносится в список при первом посещении, до посещения ее дочерних вершин.

Работа процедуры APPROX-TSP-TOUR показана на рис. 35.2. В части (а) показан полный неориентированный граф, а в части (б) — минимальное остовное дерево T с корнем a, вычисленное процедурой MST-PRIM. В части (в) показано, как прямой обход дерева T посещает вершины, а в части (г) показан соответствующий тур, возвращаемый процедурой APPROX-TSP-TOUR. В части (г) приведен оптимальный тур, который короче примерно на 23%.

Согласно результатам упр. 23.2.2 даже при простой реализации алгоритма MST-PRIM время работы алгоритма APPROX-TSP-TOUR равно $\Theta(V^2)$. Теперь покажем, что если функция стоимости в экземпляре задачи коммивояжера удовлетворяет неравенству треугольника, то алгоритм APPROX-TSP-TOUR возвращает тур, стоимость которого не более чем в два раза превышает стоимость оптимального тура.

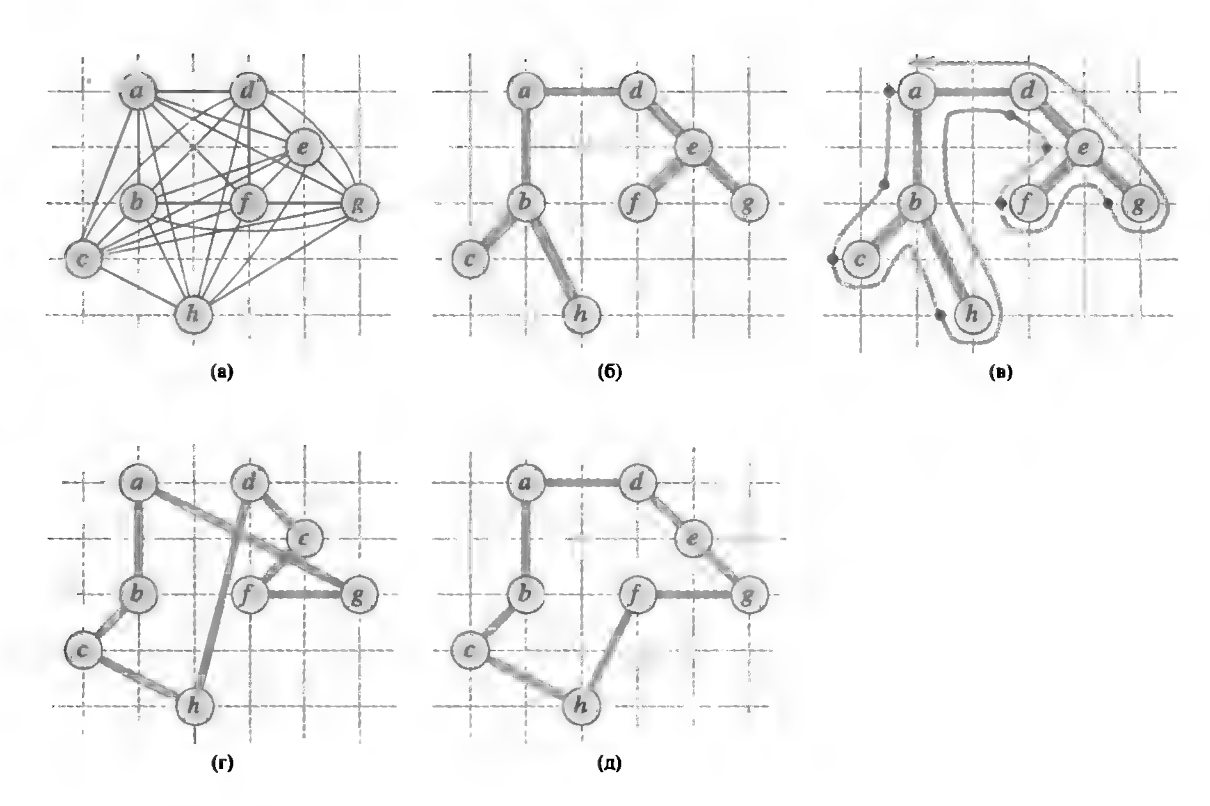


Рис. 35.2. Работа процедуры APPROX-TSP-TOUR. **(а)** Полный неориентированный граф. Вершины лежат на пересечении линий целочисленной решетки. Например, f находится на одну единицу правее и на две выше, чем h. Функция стоимости между двумя точками представляет собой обычное евклидово расстояние. **(б)** Минимальное остовное дерево T полного графа, вычисленное процедурой MST-PRIM. Корневой вершиной является вершина a. Показаны только те ребра, которые входят в минимальное остовное дерево. Метки присвоены вершинам таким образом, что они добавляются алгоритмом MST-PRIM в основное дерево в алфавитном порядке. **(в)** Порядок посещения вершин при прямом обходе дерева T, начиная с вершины a. При полном обходе дерева вершины посещаются в порядке $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. При прямом обходе дерева T составляется список вершин, посещенных впервые (на рисунке возле каждой такой вершины поставлена точка). Полученный в результате список имеет вид a, b, c, h, d, e, f, g. **(г)** Тур H, возвращенный алгоритмом APPROX-TSP-TOUR. Его полная стоимость составляет приблизительно 19.074. **(д)** Оптимальный тур H^* в исходном полном графе. Его общая стоимость приблизительно равна 14.715.

Теорема 35.2

Алгоритм APPROX-TSP-TOUR является 2-приближенным алгоритмом с полиномиальным временем работы, решающим задачу коммивояжера, в которой удовлетворяется неравенство треугольника.

Доказательство. Ранее было показано, что время работы алгоритма APPROX-TSP-TOUR выражается полиномиальной функцией.

Обозначим через H^* тур, который является оптимальным для данного множества вершин. Поскольку путем удаления из этого тура одного ребра получается остовное дерево, вес минимального остовного дерева T, вычисляемого в строке 2 процедуры APPROX-TSP-TOUR, равен нижней границе стоимости оптимального тура, т.е. выполняется неравенство

$$c(T) \le c(H^*) \ . \tag{35.4}$$

При ***полном обходе*** (full walk) дерева T составляется список вершин, которые посещаются впервые, а также когда к ним происходит возврат после посещения поддерева. Обозначим этот обход через W. При полном обходе в рассматриваемом примере вершины посещаются в следующем порядке:

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a \ .$$

Поскольку при полном обходе каждое ребро дерева T проходится ровно по два раза, естественным образом обобщив определение стоимости c на множества, в которых ребра встречаются по несколько раз, получаем равенство

$$c(W) = 2c(T) \ . \tag{35.5}$$

Из (35.4) и (35.5) вытекает, что

$$c(W) \leq 2c(H^*) \ , \tag{35.6}$$

так что стоимость обхода W превышает стоимость оптимального тура не более чем в два раза.

К сожалению, полный обход W в общем случае не является туром, поскольку он посещает некоторые вершины более одного раза. Однако согласно неравенству треугольника посещение любой из вершин в обходе W можно отменить, и при этом стоимость не возрастет. (Если из маршрута W удалить вершину v, которая посещается в этом маршруте на пути от вершины u к вершине w, то в полученном в результате такой операции упорядоченном списке вершин будет определяться переход непосредственно от вершины u к вершине w.) Путем неоднократного выполнения этой операции из обхода W можно исключить все посещения каждой вершины, кроме первого. В рассматриваемом примере упорядоченный список вершин принимает вид

$$a, b, c, h, d, e, f, g \ .$$

Этот порядок совпадает с тем, который получается при прямом обходе дерева T. Пусть H — цикл, соответствующий данному прямому обходу. Это гамильтонов цикл, так как каждая вершина посещается по одному разу. Именно этот цикл и вычисляется алгоритмом APPROX-TSP-TOUR. Поскольку цикл H получается путем удаления вершин из полного обхода W, выполняется неравенство

$$c(H) \leq c(W) \ . \tag{35.7}$$

Объединив неравенства (35.6) и (35.7), получаем $c(H) \leq 2c(H^*)$, что и завершает доказательство. ■

Несмотря на то что теорема 35.2 позволяет добиться неплохого коэффициента аппроксимации, обычно на практике алгоритм APPROX-TSP-TOUR — не лучший выбор для решения этой задачи. Существует другой приближенный алгоритм, который обычно дает намного лучшие практические результаты (см. ссылки в конце этой главы).

35.2.2. Общая задача о коммивояжере

Если отказаться от предположения о том, что функция стоимости c удовлетворяет неравенству треугольника, то нельзя найти туры с хорошим приближением за полиномиальное время, если только не выполняется условие $\text{P} = \text{NP}$.

Теорема 35.3

Если $\text{P} \neq \text{NP}$, то для любой константы $\rho \geq 1$ не существует приближенного алгоритма с полиномиальным временем работы и коэффициентом аппроксимации ρ, позволяющего решить задачу о коммивояжере в общем случае.

Доказательство. Докажем теорему "от противного". Предположим, что для некоторого числа $\rho \geq 1$ существует приближенный алгоритм A с полиномиальным временем работы и коэффициентом аппроксимации ρ. Без потери общности предположим, что число ρ — целое, при необходимости округлив его. Затем покажем, как с помощью алгоритма A можно решать экземпляры задачи о гамильтоновом цикле (определенной в разделе 34.2) за полиномиальное время. Поскольку задача о гамильтоновом цикле согласно теореме 34.13 NP-полная, из ее разрешимости за полиномиальное время и теоремы 34.4 следует равенство $\text{P} = \text{NP}$.

Пусть $G = (V, E)$ — экземпляр задачи о гамильтоновом цикле. Мы хотим эффективно определить с помощью гипотетического приближенного алгоритма A, содержит ли граф G гамильтонов цикл. Преобразуем граф G в экземпляр задачи о коммивояжере. Пусть $G' = (V, E')$ — полный граф на множестве V, т.е.

$$E' = \{(u, v) : u, v \in V \text{ и } u \neq v\} \ .$$

Назначим каждому ребру из множества E' целочисленную стоимость:

$$c(u, v) = \begin{cases} 1\ , & \text{если } (u, v) \in E\ , \\ \rho\,|V| + 1 & \text{в противном случае}\ . \end{cases}$$

Представление графа G' и функции c можно получить из представления графа G за время, полиномиально зависящее от величин $|V|$ и $|E|$.

Теперь рассмотрим задачу о коммивояжере (G', c). Если исходный граф G содержит гамильтонов цикл H, то функция стоимости c сопоставляет каждому ребру цикла H единичную стоимость, а значит, экземпляр (G', c) содержит тур стоимостью $|V|$. С другой стороны, если граф G не содержит гамильтонова цикла, то в любом туре по графу G' должно использоваться некоторое ребро, отсутствующее в множестве E. Однако стоимость любого тура, в котором используется ребро, не содержащееся в множестве E, не меньше величины

$$\begin{aligned} (\rho\,|V| + 1) + (|V| - 1) &= \rho\,|V| + |V| \\ &> \rho\,|V| \ . \end{aligned}$$

Из-за большой стоимости ребер, отсутствующих в графе G, между стоимостью тура, который представляет собой гамильтонов цикл в графе G (она равна $|V|$), и стоимостью любого другого тура (его величина не меньше $\rho\,|V|+|V|$) существу-

ет интервал, величина которого не меньше $\rho|V|$. Следовательно, стоимость тура, не являющегося гамильтоновым циклом в G, как минимум в $\rho + 1$ раз больше стоимости тура, являющегося гамильтоновым циклом в G.

Теперь предположим, что мы применяем к задаче о коммивояжере (G', c) алгоритм A. Поскольку этот алгоритм гарантированно возвращает тур, стоимость которого не более чем в ρ раз превышает стоимость оптимального тура, если граф G содержит гамильтонов цикл, алгоритм A должен его возвратить. Если же граф G не содержит гамильтоновых циклов, то алгоритм A возвращает тур, стоимость которого превышает величину $\rho|V|$. Поэтому с помощью алгоритма A задачу о гамильтоновом цикле можно решить за полиномиальное время. ■

Доказательство теоремы 35.3 служит примером общей методики доказательства того, что задачу нельзя очень хорошо аппроксимировать. Предположим, что заданной NP-сложной задаче X в течение полиномиального времени можно сопоставить такую задачу минимизации Y, что "да"-экземпляры задачи X будут соответствовать экземплярам задачи Y, стоимость которых не превышает k (где k — некоторая фиксированная величина), а "нет"-экземпляры задачи X будут соответствовать экземплярам задачи Y, стоимость которых превышает ρk. Затем мы должны показать, что, если только не выполняется равенство P = NP, не существует ρ-приближенного алгоритма с полиномиальным временем работы, позволяющего решить задачу Y.

Упражнения

35.2.1

Предположим, что полный неориентированный граф $G = (V, E)$, содержащий не менее трех вершин, характеризуется функцией стоимости c, удовлетворяющей неравенству треугольника. Докажите, что для всех $u, v \in V$ выполняется неравенство $c(u, v) \geq 0$.

35.2.2

Покажите, как за полиномиальное время один экземпляр задачи о коммивояжере можно преобразовать в другой экземпляр, функция стоимости которого удовлетворяет неравенству треугольника. Оба экземпляра должны содержать одно и то же множество оптимальных туров. Объясните, почему такое полиномиально-временное преобразование не противоречит теореме 35.3, в предположении, что P $\neq$ NP.

35.2.3

Рассмотрим описанный ниже ***эвристический метод ближайшей точки*** (closest-point heuristic), позволяющий создавать приближенные туры в задаче о коммивояжере, функция стоимости которой удовлетворяет неравенству треугольника. Начнем построение с тривиального цикла, состоящего из одной произвольным образом выбранной вершины. На каждом этапе находится вершина u, не принадлежащая циклу, причем такая, расстояние от которой до цикла является минимальным. Предположим, что ближе всех к вершине u в цикле расположена

вершина v. Цикл расширяется за счет включения в него вершины u сразу после вершины v. Описанные действия повторяются до тех пор, пока в цикл не будут включены все вершины. Докажите, что этот эвристический метод возвращает тур, полная стоимость которого не более чем в два раза превышает полную стоимость оптимального тура.

35.2.4

Задача о коммивояжере с устранением узких мест (bottleneck travelling-salesman problem) — это задача поиска такого гамильтонова цикла, для которого минимизируется стоимость самого дорогостоящего входящего в этот цикл ребра. Предполагая, что функция стоимости удовлетворяет неравенству треугольника, покажите, что для этой задачи существует приближенный алгоритм с полиномиальным временем работы, коэффициент аппроксимации которого равен 3. (*Указание:* воспользовавшись рекурсией, покажите, что все узлы остовного дерева с узкими местами можно обойти ровно по одному разу, беря полный обход дерева и выполняя в нем пропуски таким образом, чтобы не пропускать более двух последовательных промежуточных узлов (см. задачу 23.3). Покажите, что стоимость самого дорогостоящего ребра в остовном дереве с узкими местами не превышает стоимости самого дорогостоящего ребра в гамильтоновом цикле с узкими местами.)

35.2.5

Предположим, что вершины экземпляра задачи о коммивояжере расположены на плоскости и что стоимость $c(u, v)$ равна евклидову расстоянию между точками u и v. Покажите, что в оптимальном туре никогда не будет самопересечений.

35.3. Задача о покрытии множества

Задача о покрытии множества — это задача оптимизации, моделирующая многие задачи распределения ресурсов. Соответствующая ей задача принятия решения представляет собой обобщение NP-полной задачи о вершинном покрытии и, таким образом, является NP-сложной. Однако приближенный алгоритм, разработанный для задачи о вершинном покрытии, в данном случае неприменим, поэтому следует попытаться поискать другие подходы. Исследуем простой эвристический жадный метод, коэффициент аппроксимации которого выражается логарифмической функцией. Другими словами, по мере того как растет размер экземпляра задачи, размер приближенного решения также может возрастать относительно размера оптимального решения. Однако, так как логарифмическая функция возрастает достаточно медленно, этот приближенный алгоритм может давать полезные результаты.

Экземпляр $(X, \mathcal{F})$ ***задачи о покрытии множества*** (set-covering problem) состоит из конечного множества X и такого семейства $\mathcal{F}$ подмножеств множества X, что каждый элемент множества X принадлежит хотя бы одному подмножеству

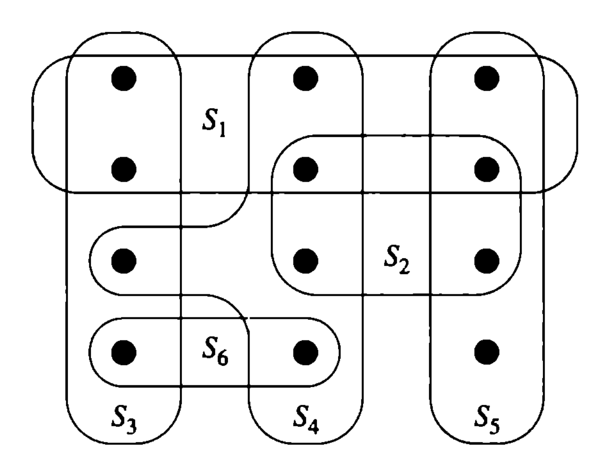


Рис. 35.3. Экземпляр $(X, \mathcal{F})$ задачи о покрытии множества, где X состоит из 12 черных точек, а $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Минимальным является покрытие $\mathcal{C} = \{S_3, S_4, S_5\}$, размер которого равен 3. Жадный алгоритм дает покрытие размером 4, выбирая либо множества S_1, S_4, S_5 и S_3, либо множества S_1, S_4, S_5 и S_6 в указанном порядке.

из семейства $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

Говорят, что подмножество $S \in \mathcal{F}$ ***покрывает*** (covers) содержащиеся в нем элементы. Задача состоит в том, чтобы найти подмножество $\mathcal{C} \subseteq \mathcal{F}$ минимального размера, члены которого покрывают все множество X:

$$X = \bigcup_{S \in \mathcal{C}} S . \qquad (35.8)$$

Говорят, что любое семейство $\mathcal{C}$, удовлетворяющее уравнению (35.8), ***покрывает*** (covers) множество X. Задача о покрытии множества иллюстрируется на рис. 35.3. Размер семейства $\mathcal{C}$ определяется как количество содержащихся в нем подмножеств, а не как суммарное количество отдельных элементов в этих множествах. В примере, проиллюстрированном на рис. 35.3, размер минимального покрытия множества равен 3.

Задача о покрытии множества абстрагирует многие часто возникающие комбинаторные задачи. В качестве простого примера предположим, что множество X представляет набор знаний, необходимых для решения задачи, над которой работает определенный коллектив сотрудников. Нужно сформировать комитет, состоящий из минимально возможного количества сотрудников, причем такой, что при необходимости получения любой информации из множества X окажется, что в комитете есть сотрудник, обладающий необходимыми знаниями. Если преобразовать эту задачу в задачу принятия решений, то в ней будет спрашиваться, существует ли покрытие, размер которого не превышает k, где k — дополнительный параметр, определенный в экземпляре задачи. Как предлагается показать в упр. 35.3.2, версия этой задачи в форме задачи принятия решений является NP-полной.

Жадный приближенный алгоритм

Жадный метод работает путем выбора на каждом этапе множества S, покрывающего максимальное количество элементов, оставшихся непокрытыми.

```
GREEDY-SET-COVER(X, F)
1  U = X
2  C = ∅
3  while U ≠ ∅
4      Выбрать S ∈ F, максимизирующее |S ∩ U|
5      U = U − S
6      C = C ∪ {S}
7  return C
```

В примере на рис. 35.3 процедура GREEDY-SET-COVER поочередно добавляет к $\mathcal{C}$ множества S_1, S_4 и S_5, после чего добавляется либо S_3, либо S_6.

Алгоритм работает следующим образом. На каждом этапе его работы множество U содержит элементы, оставшиеся непокрытыми. Множество $\mathcal{C}$ содержит покрытие, которое строится алгоритмом. Строка 4 представляет собой этап принятия решения в жадном методе. Выбирается подмножество S, покрывающее максимально возможное количество еще непокрытых элементов (с произвольным разрешением неоднозначностей). После выбора подмножества S его элементы удаляются из множества U, а само подмножество S помещается в семейство $\mathcal{C}$. Когда алгоритм завершит свою работу, множество $\mathcal{C}$ будет содержать подсемейство семейства $\mathcal{F}$, покрывающее множество X.

Алгоритм GREEDY-SET-COVER легко реализовать таким образом, чтобы время его работы выражалось полиномиальной функцией от величин $|X|$ и $|\mathcal{F}|$. Поскольку количество итераций цикла в строках 3–6 ограничено сверху величиной $\min(|X|, |\mathcal{F}|)$, а тело цикла можно реализовать таким образом, чтобы его выполнение завершалось за время $O(|X|\,|\mathcal{F}|)$, простая реализация алгоритма имеет время работы, равное $O(|X|\,|\mathcal{F}|\min(|X|, |\mathcal{F}|))$. В упр. 35.3.3 предлагается разработать алгоритм с линейным временем работы.

Анализ

Теперь покажем, что жадный алгоритм возвращает покрытие множества, не слишком сильно превышающее оптимальное покрытие. Для удобства в этой главе d-е по порядку гармоническое число $H_d = \sum_{i=1}^{d} 1/i$ (см. раздел А.1) будет обозначаться как $H(d)$. В качестве граничного условия определим $H(0) = 0$.

Теорема 35.4

Алгоритм GREEDY-SET-COVER является $\rho(n)$-приближенным алгоритмом с полиномиальным временем работы, где

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) .$$

Доказательство. Мы уже показали, что алгоритм GREEDY-SET-COVER выполняется за полиномиальное время.

Чтобы показать, что GREEDY-SET-COVER является $\rho(n)$-приближенным алгоритмом, присвоим каждому из выбранных алгоритмом множеств стоимость 1, распределим ее по всем элементам, покрытым за первый раз, а затем с помощью этих стоимостей получим искомое соотношение между размером оптимального покрытия множества $\mathcal{C}^*$ и размером покрытия $\mathcal{C}$, возвращенного алгоритмом. Обозначим i-е подмножество, выбранное алгоритмом GREEDY-SET-COVER, как S_i; добавление подмножества S_i в множество $\mathcal{C}$ приводит к увеличению стоимости на единицу. Равномерно распределим стоимость, соответствующую выбору подмножества S_i, между элементами, которые впервые покрываются этим подмножеством. Обозначим через c_x стоимость, выделенную элементу x, для каждого $x \in X$. Стоимость выделяется каждому элементу только один раз, когда этот элемент покрывается впервые. Если элемент x первый раз покрывается подмножеством S_i, то выполняется равенство

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

На каждом шаге алгоритма присваивается единичная стоимость, поэтому

$$|\mathcal{C}| = \sum_{x \in X} c_x . \tag{35.9}$$

Каждый элемент $x \in X$ находится как минимум в одном множестве из оптимального покрытия $\mathcal{C}^*$, так что

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x . \tag{35.10}$$

Объединив (35.9) и (35.10), получаем соотношение

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x . \tag{35.11}$$

Оставшаяся часть доказательства основана на приведенном ниже ключевом неравенстве, которое будет доказано чуть позже. Для любого множества S, принадлежащего семейству $\mathcal{F}$, выполняется неравенство

$$\sum_{x \in S} c_x \leq H(|S|) . \tag{35.12}$$

Из неравенств (35.11) и (35.12) следует, что

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

что и доказывает теорему.

Все, что осталось сделать — это доказать неравенство (35.12). Рассмотрим произвольные множество $S \in \mathcal{F}$ и индекс $i = 1, 2, \ldots, |\mathcal{C}|$ и введем величину

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)| \ ,$$

которая равна количеству элементов множества S, оставшихся непокрытыми после того, как в алгоритме были выбраны множества $S_1, S_2, \ldots, S_i$. Определим величину $u_0 = |S|$, равную количеству элементов множества S, которые изначально непокрыты. Пусть k — минимальный индекс, при котором выполняется равенство $u_k = 0$, т.е. каждый элемент множества S покрывается хотя бы одним из множеств $S_1, S_2, \ldots, S_k$. Тогда $u_{i-1} \geq u_i$, и при $i = 1, 2, \ldots, k$ множеством S_i впервые покрываются $u_{i-1} - u_i$ элементов множества S. Таким образом,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \ .$$

Заметим, что

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \\ &= u_{i-1} \ , \end{aligned}$$

поскольку при жадном выборе множества S_i гарантируется, что множество S не может покрыть больше новых элементов, чем множество S_i (в противном случае вместо множества S_i было бы выбрано множество S). Таким образом, мы получаем неравенство

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \ .$$

Теперь ограничим эту величину следующим образом:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && \text{(поскольку } j \leq u_{i-1}\text{)} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \end{aligned}$$

$$
\begin{aligned}
&= H(u_0) - H(u_k) && \text{(благодаря телескопичности суммы)} \\
&= H(u_0) - H(0) \\
&= H(u_0) && \text{(поскольку } H(0) = 0\text{)} \\
&= H(|S|) ,
\end{aligned}
$$

что и завершает доказательство неравенства (35.12). ■

Следствие 35.5

Алгоритм GREEDY-SET-COVER является $(\ln |X| + 1)$-приближенным алгоритмом с полиномиальным временем работы.

Доказательство. Достаточно воспользоваться неравенством (A.14) и теоремой 35.4. ■

В некоторых приложениях величина $\max\{|S| : S \in \mathcal{F}\}$ представляет собой небольшую константу, поэтому решение, которое возвращается алгоритмом GREEDY-SET-COVER, больше оптимального на множитель, не превышающий малую константу. Одно из таких приложений — получение с помощью описанного выше эвристического метода приближенного вершинного покрытия графа, степень вершин которого не превышает 3. В этом случае решение, найденное алгоритмом GREEDY-SET-COVER, не более чем в $H(3) = 11/6$ раз больше оптимального решения, т.е. оно несколько лучше решения, предоставляемого алгоритмом APPROX-VERTEX-COVER.

Упражнения

35.3.1

Будем рассматривать каждое из приведенных далее слов как множество букв: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Приведите покрытие множества, которое будет возвращено алгоритмом GREEDY-SET-COVER, если неоднозначности разрешаются в пользу слов, которые находятся в словаре раньше других.

35.3.2

Покажите, что задача о покрытии множества в форме задачи принятия решения является NP-полной. Для этого приведите к ней задачу о вершинном покрытии.

35.3.3

Покажите, как реализовать процедуру GREEDY-SET-COVER, чтобы она выполнялась за время $O\left(\sum_{S\in\mathcal{F}} |S|\right)$.

35.3.4

Покажите, что приведенное ниже неравенство (более слабое по сравнению с использованным в теореме 35.4) выполняется тривиальным образом:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} .$$

35.3.5

В зависимости от принципа разрешения неоднозначностей в строке 4 алгоритм GREEDY-SET-COVER может возвращать несколько разных решений. Разработайте процедуру BAD-SET-COVER-INSTANCE(n), возвращающую n-элементный экземпляр задачи о покрытии множества, для которого процедура GREEDY-SET-COVER при разной организации разрешения неоднозначностей в строке 4 могла бы возвращать различные решения, количество которых выражалось бы показательной функцией от n.

35.4. Рандомизация и линейное программирование

В этом разделе мы изучим два весьма полезных для разработки приближенных алгоритмов метода: рандомизацию и линейное программирование. Здесь будет приведен простой рандомизированный алгоритм, позволяющий создать оптимизирующую версию решения задачи о 3-CNF-выполнимости, после чего с помощью методов линейного программирования будет разработан приближенный алгоритм для взвешенной версии задачи о вершинном покрытии. В тексте раздела эти два мощных метода рассматриваются лишь поверхностно, но в заключительных замечаниях к главе даются ссылки для дальнейшего изучения этой темы.

Рандомизированный приближенный алгоритм для задачи о MAX-3-CNF-выполнимости

Рандомизированные алгоритмы могут применяться для поиска как точных, так и приближенных решений. Говорят, что рандомизированный алгоритм решения задачи имеет ***коэффициент аппроксимации*** (approximation ratio) $\rho(n)$, если для любых входных данных размера n *ожидаемая* стоимость C решения, полученного с помощью этого рандомизированного алгоритма, не более чем в $\rho(n)$ раз превышает стоимость C^* оптимального решения:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) . \qquad (35.13)$$

Рандомизированный алгоритм, позволяющий получить коэффициент аппроксимации $\rho(n)$, называют ***рандомизированным $\rho(n)$-приближенным алгоритмом*** (randomized $\rho(n)$-approximation algorithm). Другими словами, рандомизированный приближенный алгоритм похож на детерминистический приближенный алгоритм, с тем отличием, что его коэффициент аппроксимации относится к ожидаемой стоимости.

Конкретный экземпляр задачи о 3-CNF-выполнимости, определенной в разделе 34.4, может не выполняться. Чтобы он был выполнимым, должен существовать такой вариант присвоения переменных, при котором каждое подвыражение в скобках принимает значение 1. Если экземпляр невыполнимый, может возникнуть потребность оценить, насколько он "близок" к выполнимому. Другими сло-

вами, может возникнуть желание определить, какие значения следует присвоить переменным, чтобы выполнялось максимально возможное количество подвыражений в скобках. Назовем задачу, которая получилась в результате, задачей о ***MAX-3-CNF-выполнимости*** (MAX-3-CNF satisfiability). Входные данные этой задачи совпадают с входными данными задачи о 3-CNF-выполнимости, а цель состоит в том, чтобы найти присваиваемые переменным значения, при которых значение 1 принимает максимальное количество подвыражений в скобках. Теперь покажем, что если значения присваиваются каждой переменной случайным образом, причем значения 0 и 1 присваивается с вероятностью $1/2$, то получается рандомизированный $8/7$-приближенный алгоритм. В соответствии с определением 3-CNF-выполнимости, приведенном в разделе 34.4, требуется, чтобы в каждом подвыражении в скобках содержалось ровно три различных литерала. Кроме того, предполагается, что ни одно из выражений в скобках не содержит одновременно переменной и ее отрицания. (В упр. 35.4.1 предлагается отказаться от этого предположения.)

Теорема 35.6

Для заданного экземпляра задачи о MAX-3-CNF-выполнимости с n переменными $x_1, x_2, \ldots, x_n$ и m подвыражениями в скобках рандомизированный алгоритм, в котором каждой переменной независимо с вероятностью $1/2$ присваивается значение 1 и с той же вероятностью $1/2$ — значение 0, является рандомизированным $8/7$-приближенным алгоритмом.

Доказательство. Предположим, что каждой переменной независимо с вероятностью $1/2$ присваивается значение 1 и с той же вероятностью $1/2$ — значение 0. Определим для $i = 1, 2, \ldots, m$ индикаторную случайную величину

$$Y_i = \mathrm{I}\{\text{подвыражение } i \text{ выполняется}\} \ ,$$

так что равенство $Y_i = 1$ выполняется, если хотя бы одному из литералов, содержащихся в i-м выражении в скобках, присвоено значение 1. Поскольку ни один из литералов не входит в одно и то же подвыражение в скобках более одного раза и поскольку предполагается, что одни и те же скобки не содержат одновременно переменную и ее отрицание, присвоение значений трем переменным в каждых скобках выполняется независимым образом. Выражение в скобках не выполняется только тогда, когда всем трем его литералам присваивается значение 0, поэтому $\Pr\{\text{подвыражение } i \text{ не выполняется}\} = (1/2)^3 = 1/8$. Таким образом, мы имеем $\Pr\{\text{подвыражение } i \text{ выполняется}\} = 1 - 1/8 = 7/8$, и согласно лемме 5.1 имеем $\mathrm{E}[Y_i] = 7/8$. Пусть Y — общее количество выполняющихся подвыражений, так что $Y = Y_1 + Y_2 + \cdots + Y_m$. Тогда

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(из линейности математического ожидания)}
\end{aligned}$$

$$= \sum_{i=1}^{m} 7/8$$
$$= 7m/8 \ .$$

Очевидно, что верхняя граница количества выполняющихся подвыражений в скобках равна m, поэтому коэффициент аппроксимации не превышает значения $m/(7m/8) = 8/7$. ■

Аппроксимация взвешенного вершинного покрытия с помощью линейного программирования

В ***задаче о вершинном покрытии с минимальным весом*** (minimum-weight vertex-cover problem) задается неориентированный граф $G = (V, E)$, в котором каждой вершине $v \in V$ назначается положительный вес $w(v)$. Вес любого вершинного покрытия $V' \subseteq V$ определяется как $w(V') = \sum_{v \in V'} w(v)$. В задаче нужно найти вершинное покрытие с минимальным весом.

К этой задаче нельзя применить ни алгоритм, который использовался для поиска невзвешенного вершинного покрытия, ни рандомизированное решение, так как оба эти метода могут дать решение, далекое от оптимального. Однако с помощью задачи линейного программирования можно вычислить нижнюю границу веса, который может возникать в задаче о вершинном покрытии минимального веса. Затем мы "округлим" это решение и с его помощью получим вершинное покрытие.

Предположим, что каждой вершине $v \in V$ сопоставляется переменная $x(v)$, и поставим условие, чтобы $x(v)$ было равно либо 0, либо 1 для каждой вершины $v \in V$. Равенство $x(v) = 1$ интерпретируется как принадлежность вершины v вершинному покрытию, а равенство $x(v) = 0$ — как ее отсутствие в вершинном покрытии. Тогда ограничение, согласно которому для любого ребра (u, v) хотя бы одна из вершин u и v должна входить в вершинное покрытие, можно наложить с помощью неравенства $x(u) + x(v) \geq 1$. Такой подход приводит нас к ***0-1-целочисленной задаче линейного программирования*** (0-1 integer program) поиска вершинного покрытия с минимальным весом:

$$\text{минимизировать} \sum_{v \in V} w(v)\, x(v) \tag{35.14}$$

при условиях

$$x(u) + x(v) \geq 1 \qquad \text{для каждого } (u, v) \in E \tag{35.15}$$

$$x(v) \in \{0, 1\} \text{ для каждой } v \in V \ . \tag{35.16}$$

В частном случае, когда все веса $w(v)$ равны 1, мы получаем NP-сложную оптимизирующую версию задачи о вершинном покрытии. Предположим, однако, что ограничение $x(v) \in \{0, 1\}$ убирается, а вместо него накладывается условие $0 \leq x(v) \leq 1$. Тогда мы получим ***ослабленную задачу линейного программиро-***

вания (linear-programming relaxation):

$$\text{минимизировать} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.17}$$

при условиях

$$x(u) + x(v) \geq 1 \text{ для каждого } (u, v) \in E \tag{35.18}$$

$$x(v) \leq 1 \text{ для каждой } v \in V \tag{35.19}$$

$$x(v) \geq 0 \text{ для каждой } v \in V\ . \tag{35.20}$$

Любое допустимое решение 0-1 целочисленной задачи линейного программирования, определенной в (35.14)–(35.16), является также допустимым решением задачи линейного программирования, определенной в (35.17)–(35.20). Поэтому оптимальное решение задачи линейного программирования является нижней границей оптимального решения 0-1 целочисленной задачи, а следовательно, нижней границей оптимального решения задачи о вершинном покрытии с минимальным весом.

В приведенной ниже процедуре с помощью решения сформулированной выше задачи линейного программирования строится приближенное решение задачи о вершинном покрытии с минимальным весом.

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  Вычисление x̄, оптимального решения задачи
       линейного программирования (35.17)–(35.20)
3  for каждой v ∈ V
4      if x̄(v) ≥ 1/2
5          C = C ∪ {v}
6  return C
```

Процедура APPROX-MIN-WEIGHT-VC работает следующим образом. В строке 1 вершинное покрытие инициализируется пустым множеством. В строке 2 формулируется и решается задача линейного программирования, определенная в (35.17)–(35.20). В оптимальном решении с каждой вершиной v связано значение $0 \leq \bar{x}(v) \leq 1$. С помощью этой величины в строках 3–5 определяется, какие вершины добавятся в вершинное покрытие C. Если $\bar{x}(v) \geq 1/2$, вершина v добавляется в покрытие C; в противном случае она не добавляется. В результате каждая дробная величина, входящая в решение задачи линейного программирования, "округляется" и получается решение 0-1 целочисленной задачи, определенной в (35.14)–(35.16). Наконец в строке 6 возвращается вершинное покрытие C.

Теорема 35.7

Алгоритм APPROX-MIN-WEIGHT-VC является 2-приближенным алгоритмом с полиномиальным временем работы, позволяющим решить задачу о вершинном покрытии с минимальным весом.

Доказательство. Поскольку существует алгоритм с полиномиальным временем решения, позволяющий решить задачу линейного программирования в строке 2, и поскольку цикл **for** в строках 3–5 выполняется за полиномиальное время, алгоритм APPROX-MIN-WEIGHT-VC имеет полиномиальное время работы.

Теперь покажем, что он является 2-приближенным алгоритмом. Пусть C^* — оптимальное решение задачи о вершинном покрытии с минимальным весом, а z^* — значение оптимального решения задачи линейного программирования, определенной в (35.17)–(35.20). Поскольку оптимальное вершинное покрытие является допустимым решением этой задачи, величина z^* должна быть нижней границей величины $w(C^*)$, т.е. выполняется неравенство

$$z^* \leq w(C^*) . \tag{35.21}$$

Далее, утверждается, что при округлении дробных значений переменных $\bar{x}(v)$ получается множество C, которое является вершинным покрытием, удовлетворяющим неравенству $w(C) \leq 2z^*$. Чтобы убедиться, что C — вершинное покрытие, рассмотрим произвольное ребро $(u, v) \in E$. Согласно ограничению (35.18) должно выполняться неравенство $x(u) + x(v) \geq 1$, из которого следует, что хотя бы одна из величин $\bar{x}(u)$ и $\bar{x}(v)$ не меньше $1/2$. Поэтому хотя бы одна из вершин u и v будет включена в вершинное покрытие, а следовательно, будут покрыты все ребра.

Теперь рассмотрим, чему равен вес этого покрытия. Имеем следующую цепочку соотношений:

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) .
\end{aligned} \tag{35.22}$$

Объединение неравенств (35.21) и (35.22) дает

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

а следовательно, APPROX-MIN-WEIGHT-VC является 2-приближенным алгоритмом. ■

Упражнения

35.4.1

Покажите, что даже если бы подвыражение в скобках могло содержать одновременно переменную и ее отрицание, то в результате независимого присвоения каждой переменной значения 1 с вероятностью $1/2$ и значения 0 с той же вероятностью все равно получился бы рандомизированный $8/7$-приближенный алгоритм.

35.4.2

Задача о MAX-CNF-выполнимости (MAX-CNF satisfiability problem) похожа на задачу о MAX-3-CNF-выполнимости с тем исключением, что в каждом подвыражении в скобках необязательно содержится по три литерала. Разработайте рандомизированный 2-приближенный алгоритм, позволяющий решить задачу о MAX-CNF-выполнимости.

35.4.3

В задаче MAX-CUT задан невзвешенный неориентированный граф $G = (V, E)$. Определим разрез $(S, V - S)$, как это было сделано в главе 23, а ***вес*** (weight) этого разреза, как количество пересекающих его ребер. Требуется найти разрез с максимальным весом. Предположим, что каждая вершина v случайно и независимо с вероятностью $1/2$ помещается в множество S или $V - S$. Покажите, что этот алгоритм является рандомизированным 2-приближенным алгоритмом.

35.4.4

Покажите, что ограничение (35.19) избыточно в том смысле, что если опустить его из задачи линейного программирования, определенной в (35.17)–(35.20), то любое оптимальное решение полученной в результате задачи линейного программирования для каждой вершины $v \in V$ должно удовлетворять неравенству $x(v) \leq 1$.

35.5. Задача о сумме подмножества

Вспомним из раздела 34.5.5, что экземпляр задачи о сумме подмножества имеет вид пары (S, t), где S — множество $\{x_1, x_2, \ldots, x_n\}$ положительных целых чисел, а t — положительное целое число. В этой задаче принятия решений спрашивается, существует ли подмножество множества S, сумма элементов которого равна целевому значению t. Эта задача является NP-полной (см. раздел 34.5.5).

Задача оптимизации, связанная с этой задачей принятия решений, возникает в различных практических приложениях. В такой задаче оптимизации требуется найти подмножество множества $\{x_1, x_2, \ldots, x_n\}$, сумма элементов которого принимает максимально возможное значение, не большее t. Например, пусть имеется грузовик, грузоподъемность которого не превышает t кг, и n различных ящиков, которые нужно перевезти. Вес i-го ящика равен x_i кг. Требуется загрузить грузо-

вик максимально возможным весом, но так, чтобы не превысить его грузоподъемность.

В этом разделе приведен алгоритм, позволяющий решить задачу оптимизации за экспоненциальное время. Затем показано, как модифицировать приведенный алгоритм, чтобы он превратился в схему аппроксимации с полностью полиномиальным временем решения. (Напомним, что время работы схемы аппроксимации с полностью полиномиальным временем выполнения выражается полиномиальной функцией от величины $1/\epsilon$ и от размера входных данных.)

Точный алгоритм с экспоненциальным временем работы

Предположим, что для каждого подмножества S' множества S вычисляется сумма его элементов, после чего из всех подмножеств, сумма элементов которых не превышает t, выбирается подмножество, для которого эта сумма меньше других отличается от t. Очевидно, что такой алгоритм возвратит оптимальное решение, но время его работы выражается показательной функцией. Для реализации этого алгоритма можно было бы воспользоваться итеративной процедурой, в которой в i-й итерации вычислялись бы суммы элементов всех подмножеств множества $\{x_1, x_2, \ldots, x_i\}$ на основе вычисленных ранее сумм всех подмножеств множества $\{x_1, x_2, \ldots, x_{i-1}\}$. Если пойти этим путем, то легко понять, что нет смысла продолжать обработку некоторого подмножества S' после того, как сумма его элементов превысит t, поскольку никакое надмножество S' не может быть оптимальным решением. Реализуем эту стратегию.

В качестве входных данных процедуры EXACT-SUBSET-SUM, с псевдокодом которой мы скоро ознакомимся, выступают множество $S = \{x_1, x_2, \ldots, x_n\}$ и целевое значение t. В этой процедуре итеративно вычисляется список L_i, содержащий суммы всех подмножеств множества $\{x_1, \ldots, x_i\}$, которые не превышают t. Затем возвращается максимальное значение из списка L_n.

Если L — список положительных целых чисел, а x — еще одно положительное целое число, тогда через $L + x$ будет обозначаться список целых чисел, полученный из списка L путем увеличения каждого его элемента на величину x. Например, если $L = \langle 1, 2, 3, 5, 9 \rangle$, то $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Воспользуемся этим обозначением и для множеств, так что

$$S + x = \{s + x : s \in S\} \ .$$

Кроме того, нам понадобится вспомогательная процедура MERGE-LISTS(L, L'), которая возвращает отсортированный список, представляющий собой объединение входных отсортированных списков L и L' с исключением повторяющихся значений. Время выполнения процедуры MERGE-LISTS, как и время выполнения процедуры MERGE, используемой для сортировки слиянием и описанной в разделе 2.3.1, равно $O(|L| + |L'|)$. (Псевдокод процедуры MERGE-LISTS опущен.)

```
EXACT-SUBSET-SUM(S, t)
n = |S|
L_0 = ⟨0⟩
for i = 1 to n
    L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
    Удалить из L_i все элементы, превышающие t
return наибольший элемент в L_n
```

Рассмотрим, как работает процедура EXACT-SUBSET-SUM. Обозначим через P_i множество всех значений, которые можно получить, выбрав (возможно, пустое) подмножество множества $\{x_1, x_2, \ldots, x_i\}$ и просуммировав его элементы. Например, если $S = \{1, 4, 5\}$, то

$$P_1 = \{0, 1\} \ ,$$
$$P_2 = \{0, 1, 4, 5\} \ ,$$
$$P_3 = \{0, 1, 4, 5, 6, 9, 10\} \ .$$

Воспользовавшись тождеством

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \ , \tag{35.23}$$

методом математической индукции по i можно доказать (см. упр. 35.5.1), что L_i — отсортированный список, содержащий все элементы множества P_i, значения которых не превышают t. Поскольку длина списка L_i может достигать значения 2^i, в общем случае время выполнения алгоритма EXACT-SUBSET-SUM ведет себя как показательная функция, хотя в некоторых частных случаях, когда величина t представляет собой полином от $|S|$ или все содержащиеся в множестве S числа ограничены сверху полиномиальными величинами от $|S|$, это время также полиномиально зависит от $|S|$.

Схема аппроксимации с полностью полиномиальным временем работы

Схему аппроксимации с полностью полиномиальным временем работы, позволяющую получить приближенное решение задачи о сумме подмножеств, можно составить путем "сокращения" каждого списка L_i после его создания. Идея заключается в том, что если два значения в списке L мало отличаются одно от другого, то для получения приближенного решения нет смысла явно поддерживать оба эти значения. Точнее говоря, используется некоторый параметр сокращения δ, удовлетворяющий неравенству $0 < \delta < 1$. Чтобы ***сократить*** (trim) список L по параметру δ, нужно удалить из этого списка максимальное количество элементов таким образом, чтобы в полученном в результате этого сокращения списке L' для каждого удаленного из списка L элемента y содержался элемент z, аппроксимирующий элемент y, т.е.

$$\frac{y}{1+\delta} \le z \le y \ . \tag{35.24}$$

Можно считать, что элемент z "представляет" элемент y в новом списке L', т.е. каждый удаленный элемент y представлен элементом z, удовлетворяющим неравенству (35.24). Например, если $\delta = 0.1$ и

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle ,$$

то путем сокращения списка L можно получить список

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle ,$$

где удаленное значение 11 представлено значением 10, удаленные значения 21 и 22 представлены значением 20, а удаленное значение 24 представлено значением 23. Поскольку каждый элемент измененной версии списка является одновременно элементом исходного списка, сокращение может значительно уменьшить количество элементов, сохраняя в списке близкие (несколько меньшие) представительные значения каждого удаленного элемента.

Приведенная ниже процедура по заданным параметрам L и δ сокращает список $L = \langle y_1, y_2, \ldots, y_m \rangle$ за время $\Theta(m)$; при этом предполагается, что элементы списка L отсортированы в неубывающем порядке. На выходе процедуры получается сокращенный отсортированный список.

TRIM(L, δ)
Пусть m — длина списка L
$L' = \langle y_1 \rangle$
last $= y_1$
for $i = 2$ **to** m
    if $y_i >$ *last* $\cdot (1 + \delta)$ // $y_i \geq$ *last* (список L отсортирован)
        Добавить y_i в конец списка L'
        last $= y_i$
return L'

Элементы списка L сканируются в неубывающем порядке, и в возвращаемый список L' элемент помещается, только если это первый элемент списка L или если его нельзя представить последним помещенным в список L' элементом.

Располагая процедурой TRIM, схему аппроксимации можно построить следующим образом. В качестве входных данных в эту процедуру передается множество $S = \{x_1, x_2, \ldots, x_n\}$, состоящее из n целых чисел (расположенных в произвольном порядке), целевое значение t и "параметр аппроксимации" ϵ, где

$$0 < \epsilon < 1 . \qquad (35.25)$$

Процедура возвращает значение z, величина которого отличается от оптимального решения не более чем в $1 + \epsilon$ раз.

```
APPROX-SUBSET-SUM(S, t, ϵ)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4      L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5      L_i = TRIM(L_i, ϵ/2n)
6      Удалить из L_i все элементы, которые больше t
7  Пусть z* — наибольшее значение в L_n
8  return z*
```

В строке 2 список L_0 инициализируется таким образом, чтобы в нем содержался только элемент 0. Цикл **for** в строках 3–6 вычисляет отсортированный список L_i, содержащий соответствующим образом сокращенную версию множества P_i, из которого удалены все элементы, превышающие величину t. Поскольку список L_i создается на основе списка L_{i-1}, необходимо гарантировать, что повторное сокращение не внесет слишком большой неточности. Скоро станет понятно, что процедура APPROX-SUBSET-SUM возвращает корректное приближение, если таковое существует.

В качестве примера предположим, что имеется экземпляр задачи

$$S = \langle 104, 102, 201, 101 \rangle$$

с $t = 308$ и $\epsilon = 0.40$. Параметр сокращения δ равен $\epsilon/8 = 0.05$. Процедура APPROX-SUBSET-SUM в указанных строках вычисляет следующие значения:

$$
\begin{aligned}
&\text{Строка 2:} && L_0 = \langle 0 \rangle\ , \\[4pt]
&\text{Строка 4:} && L_1 = \langle 0, 104 \rangle\ , \\
&\text{Строка 5:} && L_1 = \langle 0, 104 \rangle\ , \\
&\text{Строка 6:} && L_1 = \langle 0, 104 \rangle\ , \\[4pt]
&\text{Строка 4:} && L_2 = \langle 0, 102, 104, 206 \rangle\ , \\
&\text{Строка 5:} && L_2 = \langle 0, 102, 206 \rangle\ , \\
&\text{Строка 6:} && L_2 = \langle 0, 102, 206 \rangle\ , \\[4pt]
&\text{Строка 4:} && L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle\ , \\
&\text{Строка 5:} && L_3 = \langle 0, 102, 201, 303, 407 \rangle\ , \\
&\text{Строка 6:} && L_3 = \langle 0, 102, 201, 303 \rangle\ , \\[4pt]
&\text{Строка 4:} && L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle\ , \\
&\text{Строка 5:} && L_4 = \langle 0, 101, 201, 302, 404 \rangle\ , \\
&\text{Строка 6:} && L_4 = \langle 0, 101, 201, 302 \rangle\ .
\end{aligned}
$$

Алгоритм возвращает значение $z^* = 302$, которое приближает оптимальное решение $307 = 104 + 102 + 101$ в пределах погрешности $\epsilon = 40\%$; фактически погрешность составляет 2%.

Теорема 35.8

Процедура APPROX-SUBSET-SUM является схемой аппроксимации с полностью полиномиальным временем выполнения, позволяющей решить задачу о сумме подмножеств.

Доказательство. В результате сокращения списка L_i в строке 5 и удаления из этого списка всех элементов, превышающих значение t, поддерживается свойство, что каждый элемент списка L_i также является элементом списка P_i. Поэтому значение z^*, которое возвращается в строке 8, на самом деле является суммой некоторого подмножества множества S. Обозначим оптимальное решение задачи о сумме подмножества $y^* \in P_n$. Тогда из строки 6 известно, что $z^* \leq y^*$. Согласно неравенству (35.1) нужно показать, что $y^*/z^* \leq 1 + \epsilon$. Необходимо также показать, что время работы этого алгоритма выражается полиномиальной функцией и от $1/\epsilon$, и от размера входных данных.

В упр. 35.5.2 предлагается для каждого элемента y из P_i, который не превышает t, показать, что существует элемент $z \in L_i$, такой, что

$$\frac{y}{(1+\epsilon/2n)^i} \leq z \leq y \; . \tag{35.26}$$

Неравенство (35.26) должно выполняться для $y^* \in P_n$, поэтому существует элемент $z \in L_n$, такой, что

$$\frac{y^*}{(1+\epsilon/2n)^n} \leq z \leq y^* \; ,$$

и, таким образом,

$$\frac{y^*}{z} \leq \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.27}$$

Поскольку существует элемент $z \in L_n$, удовлетворяющий неравенству (35.27), это неравенство должно быть справедливым для элемента z^*, который имеет самое большое значение в списке L_n, т.е.

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n \tag{35.28}$$

Теперь покажем, что $y^*/z^* \leq 1 + \epsilon$. Для этого покажем, что $(1 + \epsilon/2n)^n \leq 1 + \epsilon$. Согласно уравнению (3.14) имеем $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$ В упр. 35.5.3 предлагается показать, что

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 \; . \tag{35.29}$$

Следовательно, функция $(1 + \epsilon/2n)^n$ с ростом n увеличивается и стремится к своему пределу $e^{\epsilon/2}$, и мы имеем

$$\begin{aligned}
\left(1 + \frac{\epsilon}{2n}\right)^n &\leq e^{\epsilon/2} \\
&\leq 1 + \epsilon/2 + (\epsilon/2)^2 && \text{(согласно (3.13))} \\
&\leq 1 + \epsilon && \text{(согласно (35.25))} \; .
\end{aligned} \tag{35.30}$$

38 Зак. 3726

Объединение неравенств (35.28) и (35.30) завершает анализ коэффициента аппроксимации.

Чтобы показать, что процедура APPROX-SUBSET-SUM представляет собой схему аппроксимации с полностью полиномиальным временем выполнения, оценим границу длины списка L_i. После сокращения последовательные элементы z и z' в списке L_i должны удовлетворять соотношению $z'/z > 1+\epsilon/2n$. Другими словами, они должны отличаться не менее чем в $1+\epsilon/2n$ раз. Поэтому каждый список содержит значение 0, возможно, значение 1 и до $\lfloor \log_{1+\epsilon/2n} t \rfloor$ дополнительных значений. Количество элементов в каждом списке L_i не превышает

$$\begin{aligned}
\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\
&\leq \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 && \text{(согласно (3.17))} \\
&< \frac{3n \ln t}{\epsilon} + 2 && \text{(согласно (35.25))} .
\end{aligned}$$

Эта граница является полиномиальной функцией от размера входных данных, который равен сумме количества битов $\lg t$, необходимых для представления числа t, и количества битов, необходимых для представления множества S, которое, в свою очередь, полиномиально зависит от n и от $1/\epsilon$. Поскольку время выполнения процедуры APPROX-SUBSET-SUM выражается полиномиальной функцией от длины списка L_i, эта процедура является схемой аппроксимации с полностью полиномиальным временем выполнения. ■

Упражнения

35.5.1

Докажите уравнение (35.23). Затем покажите, что после выполнения строки 5 процедуры EXACT-SUBSET-SUM список L_i является отсортированным и содержит все элементы списка P_i, значения которых не превышают t.

35.5.2

Докажите неравенство (35.26) по индукции по i.

35.5.3

Докажите неравенство (35.29).

35.5.4

Как следовало бы модифицировать представленную в этом разделе схему аппроксимации, чтобы она позволяла найти хорошее приближение наименьшего значения суммы элементов некоторого подмножества заданного входного списка, не меньшего заданной величины t?

35.5.5

Измените процедуру APPROX-SUBSET-SUM таким образом, чтобы она возвращала также подмножество S, дающее сумму z^*.

Задачи

35.1. Расфасовка по контейнерам

Предположим, что имеется множество, состоящее из n предметов, причем размер i-го предмета s_i удовлетворяет неравенству $0 < s_i < 1$. Нужно упаковать все предметы в контейнеры единичного размера, использовав при этом минимальное количество контейнеров. Каждый контейнер вмещает произвольное количество объектов, лишь бы их суммарный размер не превышал 1[2].

а. Докажите, что задача по определению минимального количества необходимых контейнеров является NP-полной. (*Указание:* приведите к ней задачу о сумме подмножества.)

При эвристическом методе ***выбора первого подходящего*** (first-fit) по очереди выбираются все предметы, и каждый помещается в первый же контейнер, в который этот предмет может поместиться. Пусть $S = \sum_{i=1}^{n} s_i$.

б. Докажите, что оптимальное количество контейнеров, необходимых для упаковки всех предметов, не меньше $\lceil S \rceil$.

в. Докажите, что при использовании эвристического метода выбора первого подходящего все контейнеры, кроме, возможно, одного, будут заполнены не менее, чем наполовину.

г. Докажите, что количество контейнеров, которые используются в эвристическом методе выбора первого подходящего, никогда не превышает величину $\lceil 2S \rceil$.

д. Докажите, что коэффициент аппроксимации эвристического метода выбора первого подходящего равен 2.

е. Представьте эффективную реализацию эвристического метода выбора первого подходящего и проанализируйте время работы полученного алгоритма.

35.2. Приближение размера максимальной клики

Пусть $G = (V, E)$ является неориентированным графом. Определим для любого числа $k \geq 1$ граф $G^{(k)}$ как неориентированный граф $(V^{(k)}, E^{(k)})$, где $V^{(k)}$ — множество всех упорядоченных k-кортежей вершин из множества V, а $E^{(k)}$ определяется таким образом, что $(v_1, v_2, \ldots, v_k)$ смежны с $(w_1, w_2, \ldots, w_k)$ тогда и только тогда, когда для $i = 1, 2, \ldots, k$ каждая вершина v_i является смежной в графе G вершине w_i (либо когда $v_i = w_i$).

---

[2]Еще не так давно с этой задачей на практике приходилось сталкиваться множеству пользователей компьютеров, которые старались разместить как можно больше файлов на как можно меньшем количестве дискет... — *Примеч. пер.*

а. Докажите, что размер максимальной клики в графе $G^{(k)}$ равен k-й степени размера максимальной клики в графе G.

б. Докажите, что если существует приближенный алгоритм, позволяющий найти клику максимального размера и обладающий постоянным коэффициентом аппроксимации, то для решения данной задачи существует схема аппроксимации с полиномиальным временем работы.

35.3. Взвешенная задача о покрытии множества

Предположим, что задача о покрытии множества обобщается таким образом, что каждому множеству S_i из семейства $\mathcal{F}$ сопоставляется вес w_i, а вес покрытия $\mathcal{C}$ вычисляется как $\sum_{S_i \in \mathcal{C}} w_i$. Требуется найти покрытие с минимальным весом. (В разделе 35.3 рассматривается случай, когда $w_i = 1$ для всех i.)

Покажите, как естественным образом обобщить жадный эвристический подход, который применяется в задаче о покрытии множества, так, чтобы он позволял получить приближенное решение любого экземпляра взвешенной задачи о покрытии множества. Покажите, что коэффициент аппроксимации этого эвристического подхода равен $H(d)$, где d — максимальный размер любого множества S_i.

35.4. Паросочетание максимальной мощности

Вспомним, что в неориентированном графе G паросочетанием называется такое множество ребер, в котором никакие два ребра не инцидентны одной и той же вершине. Из раздела 26.3 мы узнали, как найти максимальное паросочетание в двудольном графе. В настоящей задаче будет выполняться поиск паросочетаний в неориентированных графах общего вида (т.е. в графах, которые необязательно являются двудольными).

а. ***Максимальным паросочетанием*** (maximal matching) называется паросочетание, которое не является собственным подмножеством никакого другого паросочетания. Покажите, что максимальное паросочетание необязательно совпадает с паросочетанием максимальной мощности. Для этого приведите пример неориентированного графа G, максимальное паросочетание M в котором не является паросочетанием максимальной мощности. (*Указание:* имеется граф всего лишь с четырьмя вершинами, обладающий указанным свойством.)

б. Рассмотрим неориентированный граф $G = (V, E)$. Сформулируйте жадный алгоритм поиска максимального паросочетания в графе G, время работы которого было бы равно $O(E)$.

В этой задаче внимание сосредоточивается на поиске приближенного алгоритма с полиномиальным временем работы, позволяющего найти паросочетание максимальной мощности. Время работы самого быстрого из известных на сегодняшний день алгоритмов, предназначенных для поиска паросочетания максимальной мощности, превышает линейное (хотя и является полиномиальным); рассматриваемый же здесь приближенный алгоритм завершает свою работу строго в течение линейного времени. Вы должны будете показать, что жадный алгоритм

поиска максимального паросочетания с линейным временем выполнения, разработанный в п. (б), для задачи о паросочетании максимальной мощности является 2-приближенным алгоритмом.

в. Покажите, что размер паросочетания максимальной мощности в графе G представляет собой нижнюю границу размера произвольного вершинного покрытия в этом графе.

г. Рассмотрим максимальное паросочетание M в графе $G = (V, E)$. Пусть

$$T = \{v \in V : \text{некоторое ребро в } M \text{ инцидентно } v\} \ .$$

Что можно сказать о подграфе графа G, порожденном теми вершинами графа G, которые не принадлежат T?

д. На основании результатов п. (г) сделайте вывод о том, что величина $2|M|$ равна размеру вершинного покрытия графа G.

е. Воспользовавшись результатами решения пп. (в) и (д) задачи, докажите, что сформулированный в п. (б) жадный алгоритм является 2-приближенным алгоритмом для задачи о паросочетании максимальной мощности.

35.5. Расписание работы параллельной вычислительной машины

В задаче о ***расписании работы параллельной вычислительной машины*** (parallel-machine-scheduling problem) исходные данные представляют собой набор из n заданий $J_1, J_2, \ldots, J_n$, каждое из которых характеризуется временем обработки p_k. Для выполнения этих заданий в нашем распоряжении имеется m идентичных машин $M_1, M_2, \ldots, M_m$. Любое задание может выполняться на любой машине. Требуется составить ***расписание***, в котором для каждого задания J_k следует указать машину, на которой это задание будет выполняться, и выделенный ему интервал времени. Каждое задание J_k должно непрерывно выполняться только на одной машине M_i в течение времени p_k, и в это время на машине M_i не может выполняться никакое другое задание. Обозначим через C_k ***время завершения*** (completion time) задания J_k, т.е. момент времени, когда завершается обработка задания J_k. Для каждого расписания определяется его ***момент завершения*** (makespan), равный $C_{\max} = \max_{1 \le j \le n} C_j$. В задаче нужно найти расписание с минимальным моментом завершения.

Например, предположим, что имеется две машины, M_1 и M_2, и что нужно выполнить четыре задания, J_1, J_2, J_3, J_4, для которых $p_1 = 2$, $p_2 = 12$, $p_3 = 4$ и $p_4 = 5$. Можно предложить расписание, при котором на машине M_1 сначала выполняется задание J_1, а затем — задание J_2, а на машине M_2 сначала выполняется задание J_4, а затем — задание J_3. В этом расписании $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$ и $C_{\max} = 14$. В оптимальном расписании на машине M_1 выполняется задание J_2, а на машине M_2 — задания J_1, J_3 и J_4. В этом расписании $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$ и $C_{\max} = 12$.

В данной задаче о расписании работы параллельной вычислительной машины обозначим время завершения оптимального расписания через $C^*_{\max}$.

а. Покажите, что оптимальное время завершения по величине не меньше самого большого времени обработки, т.е. что выполняется неравенство

$$C^*_{\max} \geq \max_{1 \leq k \leq n} p_k .$$

б. Покажите, что оптимальное время завершения по величине не меньше средней загрузки машин, т.е. что справедливо неравенство

$$C^*_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k .$$

Предположим, что для составления расписания параллельных вычислительных машин используется следующий жадный алгоритм: как только машина освобождается, на ней начинает выполняться очередное задание, еще не внесенное в расписание.

в. Предложите псевдокод, реализующий этот жадный алгоритм. Чему равно время работы этого алгоритма?

г. Покажите, что для расписания, которое возвращается жадным алгоритмом, выполняется неравенство

$$C_{\max} \leq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k .$$

Сделайте вывод, согласно которому этот алгоритм является 2-приближенным алгоритмом с полиномиальным временем работы.

35.6. Приближенное вычисление наибольшего остовного дерева

Пусть $G = (V, E)$ представляет собой неориентированный граф с различными весами $w(u, v)$ каждого ребра $(u, v) \in E$. Пусть для каждой вершины $v \in V$ величина $\max(v) = \text{argmax}_{(u,v) \in E} \{w(u, v)\}$ указывает ребро максимального веса, инцидентное этой вершине. Обозначим через $S_G = \{\max(v) : v \in V\}$ множество ребер с максимальным весом, инцидентных каждой вершине, и пусть T_G представляет собой остовное дерево графа G с максимальным весом, т.е. остовное дерево с наибольшим общим весом. Для любого подмножества ребер $E' \subseteq E$ определим $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

а. Приведите пример графа как минимум с четырьмя вершинами, для которого $S_G = T_G$.

б. Приведите пример графа как минимум с четырьмя вершинами, для которого $S_G \neq T_G$.

в. Докажите, что $S_G \subseteq T_G$ для любого графа G.

г. Докажите, что $w(S_G) \geq w(T_G)/2$ для любого графа G.

д. Разработайте алгоритм со временем работы $O(V + E)$ для 2-приближенного вычисления наибольшего остовного дерева.

35.7. Приближенный алгоритм решения 0-1 задачи о рюкзаке

Вспомним задачу о рюкзаке из раздела 16.2. Имеется n предметов, причем i-й предмет стоит v_i долларов и весит w_i килограммов. У нас есть рюкзак, вмещающий не более W килограммов. Добавим дополнительные предположения о том, что каждый вес w_i не превышает W и что предметы проиндексированы в невозрастающем порядке их стоимости: $v_1 \geq v_2 \geq \cdots \geq v_n$.

В 0-1-задаче о рюкзаке требуется найти подмножество предметов, общий вес которого не превышает W, а суммарная стоимость максимальна. Непрерывная задача о рюкзаке подобна 0-1-задаче, но в ней позволено брать части предметов, а не руководствоваться принципом “все или ничего”. Если мы берем долю x_i предмета i, где $0 \leq x_i \leq 1$, то мы добавляем $x_i w_i$ к весу рюкзака и получаем стоимость $x_i v_i$. Наша цель — разработать полиномиальный 2-приближенный алгоритм для решения 0-1 задачи о рюкзаке.

Чтобы разработать алгоритм с полиномиальным временем работы, рассмотрим ограниченные экземпляры 0-1-задачи о рюкзаке. Для заданного экземпляра задачи о рюкзаке I мы образуем ограниченные экземпляры I_j, где $j = 1, 2, \ldots, n$, путем удаления предметов $1, 2, \ldots, j-1$ и требования, чтобы решение включало предмет j (весь предмет j как в непрерывной задаче, так и в 0-1-задаче о рюкзаке). В экземпляре I_1 не удаляются никакие предметы. Обозначим для экземпляра I_j через P_j оптимальное решение 0-1-задачи, а через Q_j — оптимальное решение непрерывной задачи.

а. Докажите, что оптимальное решение экземпляра I 0-1-задачи о рюкзаке является одним из $\{P_1, P_2, \ldots, P_n\}$.

б. Докажите, что можно найти оптимальное решение Q_j непрерывной задачи для экземпляра I_j путем включения предмета j с последующим применением жадного алгоритма, в котором на каждом шагу мы берем максимально возможное количество невыбранного предмета из множества $\{j+1, j+2, \ldots, n\}$ с максимальной удельной стоимостью v_i/w_i.

в. Докажите, что всегда можно построить оптимальное решение Q_j непрерывной задачи для экземпляра I_j, которое включает не более одного разделенного на части предмета. То есть для всех предметов за исключением, возможно, одного мы либо полностью кладем этот предмет в рюкзак, либо полностью от него отказываемся.

г. Для заданного оптимального решения Q_j экземпляра I_j непрерывной задачи постройте решение R_j из Q_j путем удаления разделенных на части предметов из Q_j. Пусть $v(S)$ обозначает общую стоимость предметов в решении S. Докажите, что $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

д. Разработайте алгоритм с полиномиальным временем работы, который возвращает решение с наибольшей стоимостью из множества $\{R_1, R_2, \ldots, R_n\}$, и докажите, что ваш алгоритм представляет собой полиномиальный 2-приближенный алгоритм для 0-1-задачи о рюкзаке.

Заключительные замечания

Несмотря на то что методы, которые необязательно вычисляют точные решения, были известны тысячи лет назад (например, методы приближенного вычисления числа π), понятие приближенного алгоритма имеет намного более короткую историю. Заслуги по формализации концепции приближенного алгоритма с полиномиальным временем работы Хохбаум (Hochbaum) [171] приписывает Гарею (Garey), Грэхему (Graham) и Ульману (Ullman) [127], а также Джонсону (Johnson) [189]. Первый такой алгоритм часто приписывается Грэхему (Graham) [148].

Со времени публикации этой ранней работы были разработаны тысячи приближенных алгоритмов, позволяющих решать самые разнообразные задачи. По этой теме имеется большое количество литературы. Недавно вышедшие книги Осиэлло (Ausiello) и др. [25], Хохбаума [171] и Вазирани (Vazirani) [343] полностью посвящены приближенным алгоритмам. То же самое можно сказать об обзорах Шмойса (Shmoys) [313] и Клейна (Klein) и Юнга (Young) [206]. В нескольких других книгах, таких как книги Гарея (Garey) и Джонсона (Johnson) [128], а также Пападимитриу (Papadimitriou) и Штейглица (Steiglitz) [269], также значительное внимание уделяется приближенным алгоритмам. В книге Лоулера (Lawler), Ленстры (Lenstra), Ринноя Кана (Rinnooy Kan) и Шмойса (Shmoys) [224] подробно рассматриваются приближенные алгоритмы, предназначенные для решения задачи о коммивояжере.

В книге Пападимитриу (Papadimitriou) и Штейглица (Steiglitz) авторство алгоритма APPROX-VERTEX-COVER приписывается Ф. Гаврилу (F. Gavril) и М. Яннакакису (M. Yannakakis). Большое количество усилий было направлено на исследование задачи о вершинном покрытии (в книге Хохбаума (Hochbaum) [171] перечислены 16 различных приближенных алгоритмов, предназначенных для решения этой задачи), однако значение всех коэффициентов аппроксимации не меньше $2 - o(1)$.

Алгоритм APPROX-TSP-TOUR был предложен в статье Розенкранца (Rosenkrantz), Стирнса (Stearns) и Льюиса (Lewis) [296]. Кристофидис (Christofides) усовершенствовал этот алгоритм и предложил 3/2-приближенный алгоритм, позволяющий решить задачу о коммивояжере с неравенством треугольника. Арора (Arora) [22] и Митчелл (Mitchell) [255] показали, что если точки находятся на евклидовой плоскости, то существует схема аппроксимации с полиномиальным временем работы. Теорема 35.3 доказана Сани (Sahni) и Гонзалезом (Gonzalez) [299].

Анализ жадного эвристического подхода к задаче о покрытии множества построен по аналогии с доказательством более общего результата, опубликованным

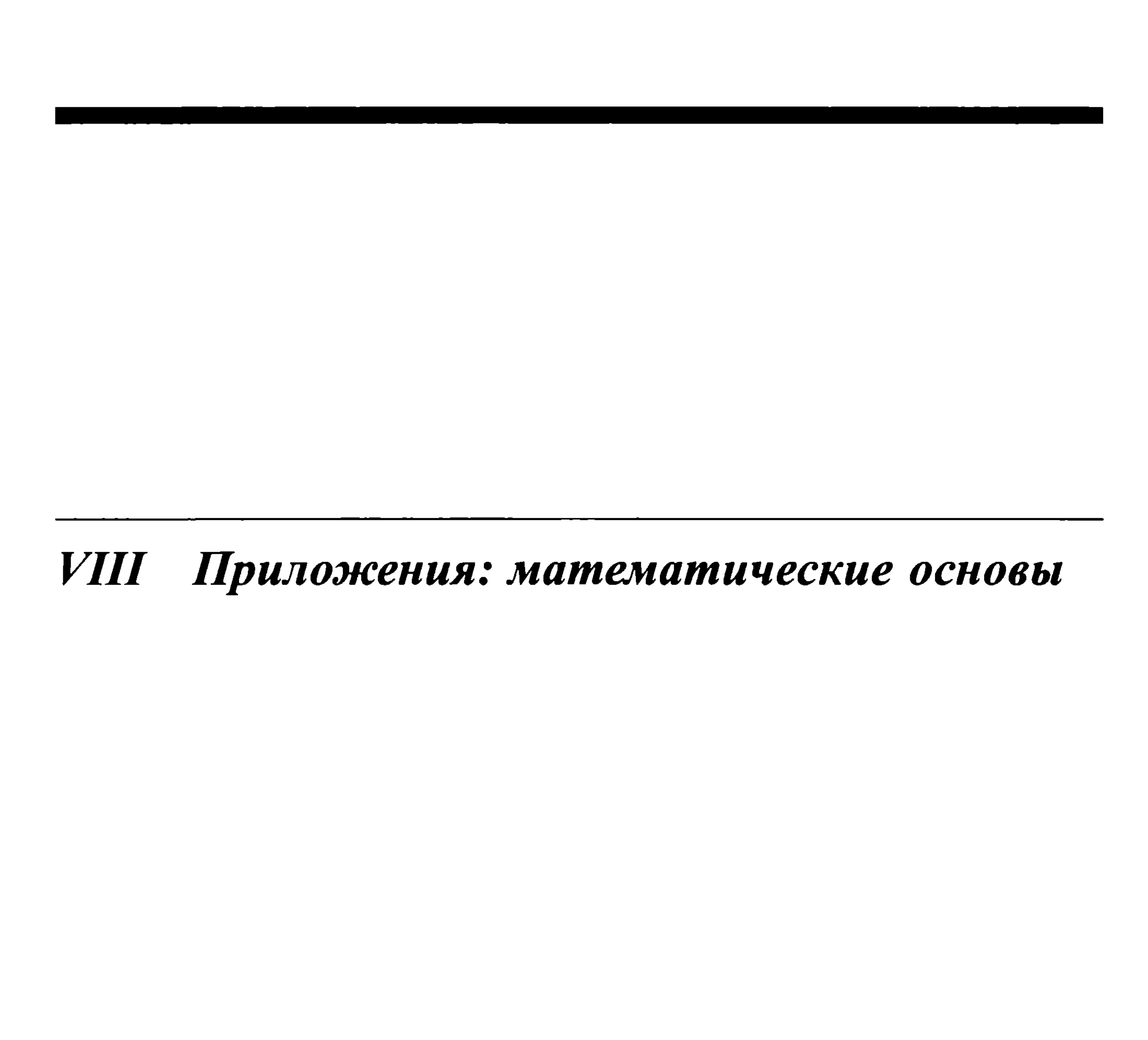
в статье Чватала (Chvátal) [67]; представленный здесь основной результат доказан Джонсоном (Johnson) [189] и Ловасом (Lovász) [237].

Описание алгоритма APPROX-SUBSET-SUM и его анализ с некоторыми изменениями взяты из статьи Ибарры (Ibarra) и Кима (Kim) [186], где приведены приближенные алгоритмы, предназначенные для решения задач о рюкзаке и о сумме подмножества.

Задача 35.7 представляет собой комбинаторную версию более общего результата по приближению целочисленной задачи о рюкзаке, полученного Бинстоком (Bienstock) и Мак-Клоски (McClosky) [44].

Рандомизированный алгоритм решения задачи о MAX-3-CNF-выполнимости можно найти в неявном виде в работе Джонсона (Johnson) [189]. Автором алгоритма, предназначенного для решения задачи о взвешенном вершинном покрытии, является Хохбаум (Hochbaum) [170]. Раздел 35.4 дает лишь поверхностное представление о тех возможностях, которые открываются благодаря использованию рандомизации и линейного программирования при разработке приближенных алгоритмов. Сочетание этих двух идей привело к появлению метода под названием "рандомизированное округление", в котором задача сначала формулируется как целочисленная задача линейного программирования. После этого решается ослабленный вариант задачи, а переменные в этом решении интерпретируются как вероятности. Затем эти вероятности используются для решения исходной задачи. Впервые этот метод был предложен Рагаваном (Raghavan) и Томсоном (Thompson) [288], после чего он нашел широкое применение. (Для ознакомления с этой темой см. обзорную статью Мотвани (Motwani), Наора (Naor) и Рагавана (Raghavan) [259].) К другим заслуживающим внимания идеям в этой области, предложенным в последнее время, относятся метод прямой двойственности (primal dual) (см. обзор Гоманса (Goemans) и Вильямсона (Williamson) [134]), поиск разреженных разрезов (sparse cuts) для использования в алгоритмах разбиения [228], а также применение полуопределенного программирования [133].

Как упоминалось в заключительных замечаниях к главе 34, последние достижения в области вероятностно проверяемых доказательств позволяют найти нижние границы аппроксимируемости многих задач, в том числе некоторых из тех, которые рассматриваются в настоящей главе. В дополнение к приведенным здесь ссылкам заметим, что глава из книги Ароры (Arora) и Ланда (Lund) [23] содержит хорошее описание отношения между вероятностно проверяемыми доказательствами и степенью сложности приближенных алгоритмов решения различных задач.



VIII Приложения: математические основы

Введение

Анализ алгоритмов требует серьезного математического аппарата. Иногда достаточно знаний из простейшего курса высшей математики, но зачастую используемые в данной книге математические концепции и методы могут оказаться новыми для вас. В части I вы уже познакомились с асимптотическими обозначениями и решением рекуррентных соотношений; в этой части вы найдете ряд других математических концепций и методов, используемых в ходе анализа алгоритмов. Как упоминалось во введении к части I, вы могли быть знакомы со многими рассматриваемыми здесь вопросами еще до того, как приступили к чтению данной книги, так что материал, представленный в приложениях, следует рассматривать, в первую очередь, как справочный. Тем не менее здесь, как и в основных главах, приведены упражнения и задачи, которые помогут вам повысить свою квалификацию в рассматриваемых областях математики.

В приложении А рассматриваются методы вычисления и оценки рядов, часто встречающихся в процессе анализа тех или иных алгоритмов. Многие из приводимых здесь формул можно найти в различных учебниках по математике, но гораздо удобнее, когда все эти формулы собраны в одном месте.

В приложении Б приведены основные определения и обозначения, используемые при работе с множествами, отношениями, функциями, графами и деревьями. В нем вы также найдете некоторые основные свойства этих математических объектов.

Приложение В начинается с элементарных принципов комбинаторики — перестановок, сочетаний и т.п. Остальной материал приложения посвящен основам теории вероятности. Большинство алгоритмов в этой книге не требуют использования теории вероятности в ходе анализа, так что можете пропустить эту часть приложения. Вы сможете вернуться к нему позже, при желании детально разобраться в вероятностном анализе алгоритмов. В этом случае вы убедитесь, что данное приложение можно рассматривать и как хорошо организованный справочник.

Приложение Г посвящено матрицам, операциям с ними и некоторым основным свойствам матриц. Вероятно, вы уже сталкивались с большей частью представленного здесь материала при прослушивании курса линейной алгебры, но может оказаться очень удобным иметь такой раздел, в который всегда можно заглянуть, чтобы освежить свою память.

Приложение А. Суммы и ряды

Если алгоритм содержит итеративную управляющую конструкцию, такую как цикл **while** или **for**, время его работы можно выразить в виде суммы значений времени выполнения отдельных итераций. Например, в разделе 2.2 указывалось, что j-я итерация алгоритма сортировки вставкой выполняется за время, в худшем случае пропорциональное j. Суммируя значения времени, затраченного на выполнение отдельных итераций, мы получим сумму, или ряд,

$$\sum_{j=2}^{n} j \; .$$

Вычисление этой суммы приводит нас к границе для времени работы алгоритма, равной $\Theta(n^2)$ в худшем случае. Этот пример свидетельствует о важности понимания и умения вычислять суммы и оценивать их границы.

В разделе А.1 перечислены некоторые основные формулы, связанные с суммированием, а в разделе А.2 — некоторые полезные методы оценки сумм. Формулы в разделе А.1 приведены без доказательств, однако в разделе А.2 в качестве иллюстрации описываемые здесь методы использованы для доказательства некоторых формул из раздела А.1. Остальные доказательства можно найти в различных учебниках по математике.

А.1. Суммы и их свойства

Для данной последовательности чисел $a_1, a_2, \ldots, a_n$, где n — неотрицательное целое число, конечная сумма $a_1 + a_2 + \cdots + a_n$ кратко записывается как

$$\sum_{k=1}^{n} a_k \; .$$

Если $n = 0$, значение суммы считается равным 0. Значение конечного ряда всегда определено и не зависит от порядка слагаемых.

Для заданной последовательности чисел $a_1, a_2, \ldots$ бесконечная сумма $a_1 + a_2 + \cdots$ кратко записывается как

$$\sum_{k=1}^{\infty} a_k \ ,$$

что рассматривается как

$$\lim_{n \to \infty} \sum_{k=1}^{n} a_k \ .$$

Если данный предел не существует, ряд ***расходится*** (diverges); в противном случае ряд ***сходится*** (converges). Члены сходящегося ряда не могут быть суммированы в произвольном порядке. Однако можно переставлять члены ***абсолютно сходящегося ряда***, т.е. ряда $\sum_{k=1}^{\infty} a_k$, для которого ряд $\sum_{k=1}^{\infty} |a_k|$ также является сходящимся.

Линейность

Для любого действительного числа c и любых конечных последовательностей $a_1, a_2, \ldots, a_n$ и $b_1, b_2, \ldots, b_n$

$$\sum_{k=1}^{n} (ca_k + b_k) = c \sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \ .$$

Свойство линейности справедливо также для бесконечных сходящихся рядов.

Это свойство может использоваться при работе с суммами, в которые входят асимптотические обозначения, например

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right) \ .$$

В этом уравнении Θ-обозначение в левой части применяется к переменной k, а в правой — к n. Аналогичные действия применимы и к бесконечным сходящимся рядам.

Арифметическая прогрессия

Сумма

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

называется ***арифметической прогрессией*** и равна

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \qquad \text{(A.1)}$$

$$= \Theta(n^2) \ . \qquad \text{(A.2)}$$

Суммы квадратов и кубов

Для сумм квадратов и кубов справедливы следующие формулы:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} , \tag{A.3}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} . \tag{A.4}$$

Геометрическая прогрессия

Для действительного $x \neq 1$ сумма

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

называется ***геометрической прогрессией*** и равна

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \tag{A.5}$$

В случае бесконечного ряда и $|x| < 1$ получается бесконечно убывающая геометрическая прогрессия, сумма которой равна

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} . \tag{A.6}$$

Поскольку мы считаем, что $0^0 = 1$, эти формулы применимы даже при $x = 0$.

Гармонический ряд

Для положительного целого n n-е ***гармоническое число*** представляет собой

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \ln n + O(1) . \end{aligned} \tag{A.7}$$

(Мы докажем это соотношение в разделе A.2.)

Интегрирование и дифференцирование рядов

Путем интегрирования или дифференцирования приведенных выше формул могут быть получены новые формулы. Например, дифференцируя обе части уравнения суммы бесконечной геометрической прогрессии (A.6) и умножая на x, мы

получим для $|x| < 1$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} . \tag{A.8}$$

Суммы разностей (телескопические ряды)

Для любой последовательности $a_0, a_1, \ldots, a_n$ справедливо соотношение

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 , \tag{A.9}$$

поскольку каждый из членов $a_1, a_2, \ldots, a_{n-1}$ прибавляется и вычитается ровно один раз. Такие ряды именуют ***телескопическими*** (telescopes). Аналогично

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n .$$

В качестве примера такого ряда рассмотрим сумму

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} .$$

Поскольку каждый ее член можно записать как

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1} ,$$

получаем

$$\begin{aligned} \sum_{k=1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) \\ &= 1 - \frac{1}{n} . \end{aligned}$$

Произведения

Конечное произведение $a_1 a_2 \cdots a_n$ может быть кратко записано следующим образом:

$$\prod_{k=1}^{n} a_k .$$

Если $n = 0$, значение произведения считается равным 1. Мы можем преобразовать формулу с произведением в формулу с суммой с помощью тождества

$$\lg \left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k .$$

Упражнения

A.1.1

Найдите простое выражение для $\sum_{k=1}^{n}(2k-1)$.

A.1.2 ★

Покажите, используя формулу для гармонического ряда, что $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$.

A.1.3

Покажите, что для $|x| < 1$ справедливо соотношение $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$.

A.1.4 ★

Покажите, что $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

A.1.5 ★

Вычислите сумму $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ для $|x| < 1$.

A.1.6

Докажите, используя свойство линейности суммирования, что $\sum_{k=1}^{n} O(f_k(i)) = O\left(\sum_{k=1}^{n} f_k(i)\right)$.

A.1.7

Вычислите произведение $\prod_{k=1}^{n} 2 \cdot 4^k$.

A.1.8 ★

Вычислите произведение $\prod_{k=2}^{n}(1 - 1/k^2)$.

А.2. Оценки сумм

Имеется множество методов оценки величин сумм, которые описывают время работы того или иного алгоритма. Здесь мы рассмотрим только наиболее распространенные из них.

Математическая индукция

Основным способом вычисления сумм рядов является метод математической индукции. В качестве примера докажем, что сумма арифметической прогрессии $\sum_{k=1}^{n} k$ равна $\frac{1}{2}n(n+1)$. Можно легко убедиться, что эта формула верна при $n = 1$. Сделаем предположение, что эта формула верна для некоторого n, и дока-

жем, что в этом случае она верна и для $n + 1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k \,+\, (n+1)$$
$$= \frac{1}{2}n(n+1) + (n+1)$$
$$= \frac{1}{2}(n+1)(n+2)\ .$$

Чтобы воспользоваться математической индукцией, не всегда требуется предположение о точном значении суммы. Индукцию можно применять и для получения границы суммы. В качестве примера докажем, что сумма геометрической прогрессии $\sum_{k=0}^{n} 3^k$ равна $O(3^n)$. Точнее говоря, докажем, что $\sum_{k=0}^{n} 3^k \leq c3^n$ для некоторой константы c. В качестве начального условия при $n = 0$ мы имеем $\sum_{k=0}^{0} 3^k = 1 \leq c \cdot 1$, что справедливо при $c \geq 1$. Считая, что указанная граница справедлива для n, докажем, что она справедлива для $n + 1$. Мы имеем

$$\begin{aligned}
\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1} \\
&\leq c3^n + 3^{n+1} && \text{(согласно гипотезе индукции)} \\
&= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \\
&\leq c3^{n+1}\ ,
\end{aligned}$$

что справедливо при $(1/3 + 1/c) \leq 1$, или, что то же самое, при $c \geq 3/2$. Таким образом, $\sum_{k=0}^{n} 3^k = O(3^n)$, что и требовалось доказать.

При использовании асимптотических обозначений для доказательства по индукции следует быть предельно внимательным и осторожным. Рассмотрим следующее “доказательство” того, что $\sum_{k=1}^{n} k = O(n)$. Очевидно, что $\sum_{k=1}^{1} k = O(1)$. Исходя из справедливости оценки для n, докажем ее для $n + 1$:

$$\begin{aligned}
\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\
&= O(n) + (n+1) && \Longleftarrow \textit{Ошибка!!} \\
&= O(n+1)\ .
\end{aligned}$$

Ошибка в том, что “константа”, скрытая в $O(n)$, растет вместе с n, а значит, константой не является. Мы не смогли показать, что одна и та же константа работает для *всех* n.

Почленное сравнение

Иногда можно получить неплохую верхнюю оценку ряда, заменив каждый его член бо́льшим (иногда для этого можно воспользоваться наибольшим членом).

Например, вот как можно оценить верхнюю границу арифметической прогрессии (A.1):

$$\sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n$$
$$= n^2 .$$

В общем случае для суммы $\sum_{k=1}^{n} a_k$, если положить $a_{\max} = \max_{1 \leq k \leq n} a_k$,

$$\sum_{k=1}^{n} a_k \leq n \cdot a_{\max} .$$

Способ ограничения наибольшим членом — весьма слабый метод, если в действительности ряд можно ограничить геометрической прогрессией. Предположим, что для ряда $\sum_{k=0}^{n} a_k$ для всех $k \geq 0$ выполняется $a_{k+1}/a_k \leq r$, где $0 < r < 1$ — константа. Мы можем ограничить сумму бесконечно убывающей геометрической прогрессией, поскольку $a_k \leq a_0 r^k$, и, таким образом

$$\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{\infty} a_0 r^k$$
$$= a_0 \sum_{k=0}^{\infty} r^k$$
$$= a_0 \frac{1}{1-r} .$$

Можно применить этот метод к ряду $\sum_{k=1}^{\infty}(k/3^k)$. Для того чтобы сумма начиналась с $k = 0$, перепишем ряд как $\sum_{k=0}^{\infty}((k+1)/3^{k+1})$. Первый член ($a_0$) равен $1/3$, а отношение последовательных членов (r) равно

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1}$$
$$\leq \frac{2}{3}$$

для всех $k \geq 0$. Таким образом, имеем

$$\sum_{k=1}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}}$$
$$\leq \frac{1}{3} \cdot \frac{1}{1-2/3}$$
$$= 1 .$$

Распространенной ошибкой при использовании этого метода является то, что из того, что отношение двух последовательных членов ряда меньше 1, делается вывод об ограниченности ряда геометрической прогрессией. В качестве контрпримера можно привести гармонический ряд, который расходится, так как

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{1}{k} &= \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} \\ &= \lim_{n\to\infty} \Theta(\lg n) \\ &= \infty \ .\end{aligned}$$

Несмотря на то что отношение $(k+1)$- и k-го членов этого ряда равно $k/(k+1) < 1$, ряд расходится. Для того чтобы ряд был ограничен геометрической прогрессией, нужно показать наличие *константы* $r < 1$, такой, что отношение двух соседних членов никогда не превосходит значения r. В случае гармонического ряда такой константы не существует, поскольку отношение двух соседних членов может быть сколь угодно близким к 1.

Разбиение сумм

Еще один способ получения оценки сложной суммы состоит в представлении ряда в виде суммы двух или более рядов путем разделения на части всего диапазона индексов и в оценке сумм частей по отдельности. Предположим, например, что мы ищем нижнюю границу арифметической прогрессии $\sum_{k=1}^{n} k$, для которой, как мы уже выяснили, верхняя граница равна n^2. Можно попытаться ограничить каждый член суммы наименьшим, но поскольку наименьший член этого ряда равен 1, мы получим в качестве нижней границы n, что слишком далеко от найденной верхней границы n^2.

Лучший результат для нижней границы можно получить, если сначала разбить ряд на две части. Предположим для удобства, что n — четное число. Тогда

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \\ &= (n/2)^2 \\ &= \Omega(n^2) \ ,\end{aligned}$$

что является асимптотически точной оценкой, поскольку $\sum_{k=1}^{n} k = O(n^2)$.

В рядах, рассматриваемых в ходе анализа алгоритмов, зачастую можно разбить ряд на части и просто проигнорировать некоторое конечное число начальных членов. Обычно этот метод применим, если каждый член a_k ряда $\sum_{k=0}^{n} a_k$ не

зависит от n. Тогда для любой константы $k_0 > 0$ можно записать

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k = \Theta(1) + \sum_{k=k_0}^{n} a_k \ ,$$

поскольку начальные члены суммы — постоянные величины, и в отдельный ряд выделено постоянное их количество. После такого разделения для поиска границ ряда $\sum_{k=k_0}^{n} a_k$ можно использовать другие методы. Описанная методика применима и к бесконечным рядам. Например, для поиска асимптотической верхней границы ряда

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$$

заметим, что отношение последовательных членов ряда

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \leq \frac{8}{9} \ ,$$

если $k \geq 3$. С учетом сказанного разобьем ряд следующим образом.

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \leq \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1) \ ,$$

поскольку количество членов первой суммы — константа, а вторая представляет собой бесконечно убывающую геометрическую прогрессию.

Метод разбиения ряда может быть применен для определения асимптотических границ в гораздо более сложных ситуациях. Например, таким способом можно получить границу $O(\lg n)$ гармонического ряда (А.7):

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \ .$$

Идея заключается в разбиении диапазона от 1 до n на $\lfloor \lg n \rfloor + 1$ частей с ограничением каждой части единицей. i-я часть ($i = 0, 1, \ldots, \lfloor \lg n \rfloor$) состоит из членов, начинающихся с $1/2^i$ и заканчивающихся членом $1/2^{i+1}$ (исключая сам этот член). Последняя часть может содержать члены, не входящие в исходный гармо-

нический ряд, и, таким образом, имеем

$$\begin{aligned}\sum_{k=1}^{n}\frac{1}{k} &\le \sum_{i=0}^{\lfloor\lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i+j}\\ &\le \sum_{i=0}^{\lfloor\lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i}\\ &= \sum_{i=0}^{\lfloor\lg n\rfloor}1\\ &\le \lg n+1\ .\end{aligned} \tag{A.10}$$

Приближение интегралами

Если сумма имеет вид $\sum_{k=m}^{n} f(k)$, где $f(k)$ — монотонно возрастающая функция, можно оценить значение ряда с помощью интегралов:

$$\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx\ . \tag{A.11}$$

Пояснение такой оценки показано на рис. А.1. На рисунке в прямоугольниках показаны их площади, а общая площадь прямоугольников представляет значение суммы. Когда $f(k)$ является монотонно убывающей функцией, можно использовать подобный метод для получения границ

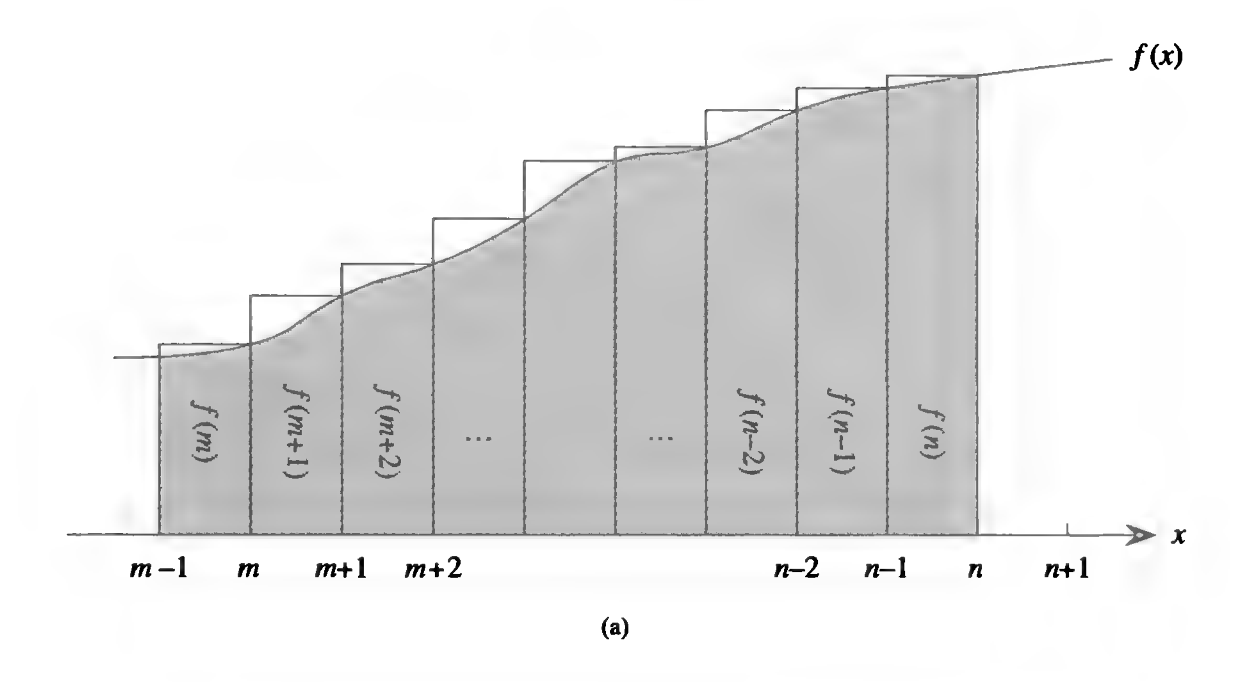
$$\int_{m}^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx\ . \tag{A.12}$$

Приближение интегралами (А.12) дает точную оценку n-го гармонического числа. Нижнюю границу получаем как

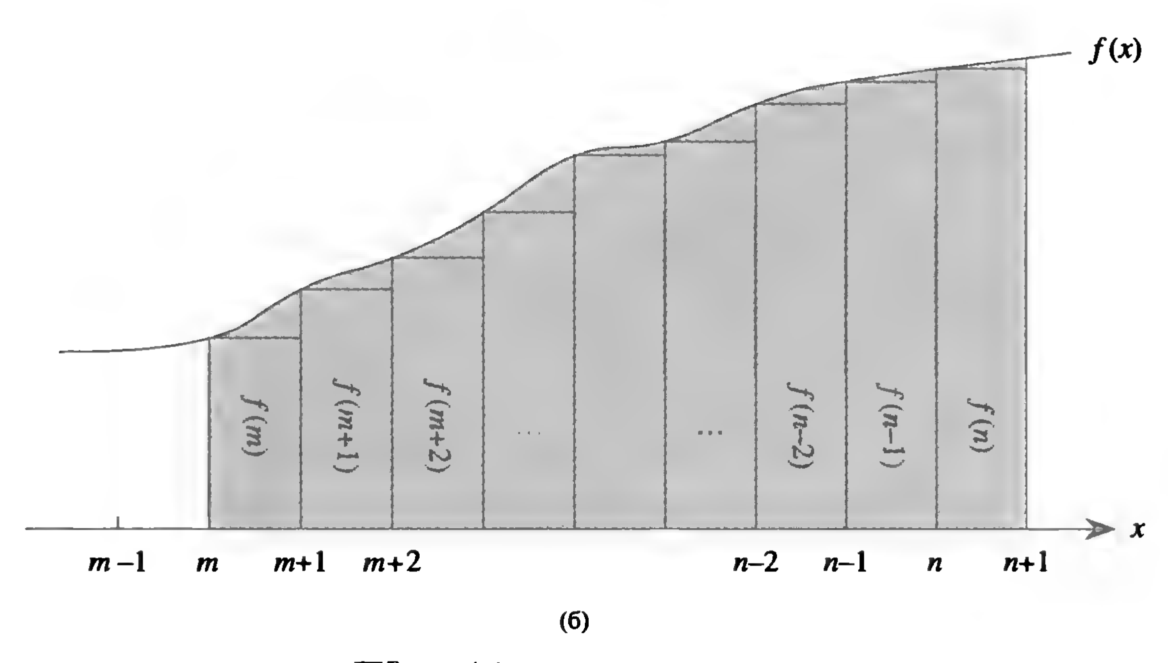
$$\begin{aligned}\sum_{k=1}^{n}\frac{1}{k} &\ge \int_{1}^{n+1}\frac{dx}{x}\\ &= \ln(n+1)\ .\end{aligned} \tag{A.13}$$

Чтобы получить верхнюю границу, воспользуемся неравенством

$$\begin{aligned}\sum_{k=2}^{n}\frac{1}{k} &\le \int_{1}^{n}\frac{dx}{x}\\ &= \ln n\ ,\end{aligned}$$



(а)



(б)

Рис. А.1. Приближение ряда $\sum_{k=m}^{n} f(k)$ интегралами. В прямоугольниках показаны их площади, а общая площадь прямоугольников представляет значение суммы. Интеграл представлен заштрихованной областью под кривой. Сравнивая площади в части **(а)**, получаем $\int_{m-1}^{n} f(x)\,dx \leq \sum_{k=m}^{n} f(k)$, после чего, сдвигая прямоугольники на единицу вправо, в части **(б)** получаем $\sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)\,dx$.

которое дает границу

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \ . \tag{A.14}$$

Упражнения

А.2.1

Покажите, что сумма $\sum_{k=1}^{n} 1/k^2$ ограничена сверху константой.

А.2.2

Найдите асимптотическую верхнюю границу суммы

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil \ .$$

А.2.3

Применив разбиение ряда, покажите, что n-е гармоническое число представляет собой $\Omega(\lg n)$.

А.2.4

Найдите приближенное значение $\sum_{k=1}^{n} k^3$ с помощью интегралов.

А.2.5

Почему для получения n-го гармонического числа мы не можем применить интегральное приближение (А.12) непосредственно к сумме $\sum_{k=1}^{n} 1/k$?

Задачи

А.1. Оценки сумм

Дайте асимптотически точные оценки приведенных ниже сумм. Считаем, что $r \geq 0$ и $s \geq 0$ представляют собой константы.

а. $\displaystyle\sum_{k=1}^{n} k^r.$

б. $\displaystyle\sum_{k=1}^{n} \lg^s k.$

в. $\displaystyle\sum_{k=1}^{n} k^r \lg^s k.$

Заключительные замечания

Книга Кнута (Knuth) [208][1] — отличное справочное пособие по изложенному здесь материалу. Основные свойства рядов можно найти во множестве книг, например в книге Апостола (Apostol) [18] или Томаса (Thomas) и др. [332].

---

[1]Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000.

Приложение Б. Множества и прочие художества

Во многих главах этой книги нам приходилось сталкиваться с элементами дискретной математики. Здесь вы более подробно ознакомитесь с обозначениями, определениями и элементарными свойствами множеств, отношений, функций, графов и деревьев. Читатели, знакомые с этим материалом, могут пропустить данное приложение.

Б.1. Множества

Множество (set) представляет собой набор различимых объектов, которые называются ***членами*** или ***элементами***. Если объект x является членом множества S, это записывается как $x \in S$ (читается "x принадлежит S"). Если x не принадлежит S, записываем $x \notin S$. Множество можно описать путем явного перечисления его элементов в виде списка, заключенного в фигурные скобки. Например, можно определить множество S как содержащее числа 1, 2 и 3, и только их, записав $S = \{1, 2, 3\}$. Поскольку 2 является элементом множества S, можно записать $2 \in S$, а так как 4 не является элементом S, верна запись $4 \notin S$. Множество не может содержать один и тот же элемент дважды[1]; кроме того, элементы множества не упорядочены. Два множества, A и B, ***равны*** (что записывается как $A = B$), если они содержат одни и те же элементы. Например, $\{1, 2, 3\} = \{3, 2, 1\}$.

Для часто встречающихся множеств используются специальные обозначения.

- $\emptyset$ обозначает ***пустое множество***, т.е. множество, не содержащее ни одного элемента.
- $\mathbb{Z}$ обозначает множество ***целых чисел***, т.е. множество $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbb{R}$ обозначает множество ***действительных чисел***.

---

[1]Модификация множества, которое может содержать несколько одинаковых элементов, называется ***мультимножеством*** (multiset).

- $\mathbb{N}$ обозначает множество ***натуральных чисел***, т.е. множество $\{1, 2, \dots\}$[2].

Если все элементы множества A содержатся в множестве B, т.е. из $x \in A$ вытекает $x \in B$, то мы записываем, что $A \subseteq B$, и говорим, что множество A является ***подмножеством*** B. Множество A является ***истинным*** (собственным, строгим) ***подмножеством*** (proper subset) множества B (что записывается как $A \subset B$), если $A \subseteq B$, но $A \neq B$. (Многие авторы используют обозначение $A \subset B$ не только для истинных подмножеств, но и для отношения обычного подмножества.) Для любого множества A справедливо $A \subseteq A$; для двух множеств, A и B, равенство $A = B$ справедливо тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$. Для любых трех множеств, A, B и C, из $A \subseteq B$ и $B \subseteq C$ следует $A \subseteq C$. Для любого множества A справедливо соотношение $\emptyset \subseteq A$.

Иногда множество определяется посредством другого множества. Так, для заданного множества A, можно определить множество $B \subseteq A$, указав свойство, которым обладают элементы множества B. Например, множество четных чисел можно определить следующим образом: $\{x : x \in \mathbb{Z} \text{ и } x/2 \in \mathbb{Z}\}$. Двоеточие в такой записи читается как "такой, что" (некоторые авторы вместо двоеточия используют вертикальную черту).

Для двух заданных множеств, A и B, можно определить новые множества с помощью следующих ***операций над множествами***.

- ***Пересечением*** множеств A и B является множество

$$A \cap B = \{x : x \in A \text{ и } x \in B\} \ .$$

- ***Объединением*** множеств A и B является множество

$$A \cup B = \{x : x \in A \text{ или } x \in B\} \ .$$

- ***Разностью*** множеств A и B является множество

$$A - B = \{x : x \in A \text{ и } x \notin B\} \ .$$

Операции над множествами обладают следующими свойствами.

Свойства пустого множества:

$$\begin{aligned} A \cap \emptyset &= \emptyset \ , \\ A \cup \emptyset &= A \ . \end{aligned}$$

---

[2]В зарубежной литературе (в частности, в американской) под натуральными числами подразумеваются числа $\{0, 1, 2, \dots\}$. Здесь мы придерживаемся принятого в отечественной математике понятия натуральных чисел. — *Примеч. ред.*

Свойства идемпотентности:

$$A \cap A = A \,,$$
$$A \cup A = A \,.$$

Свойства коммутативности:

$$A \cap B = B \cap A \,,$$
$$A \cup B = B \cup A \,.$$

Свойства ассоциативности:

$$A \cap (B \cap C) = (A \cap B) \cap C \,,$$
$$A \cup (B \cup C) = (A \cup B) \cup C \,.$$

Свойства дистрибутивности:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \,, \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \,. \end{aligned} \tag{Б.1}$$

Свойства поглощения:

$$A \cap (A \cup B) = A \,,$$
$$A \cup (A \cap B) = A \,.$$

Законы де Моргана:

$$\begin{aligned} A - (B \cap C) &= (A - B) \cup (A - C) \,, \\ A - (B \cup C) &= (A - B) \cap (A - C) \,. \end{aligned} \tag{Б.2}$$

На рис. Б.1 первый закон де Моргана проиллюстрирован с помощью ***диаграмм Венна*** (Venn diagram), графического представления множеств в виде областей на плоскости.

Зачастую все рассматриваемые множества являются подмножествами некоторого большего множества U, называемого ***универсумом*** (universe), или генеральной совокупностью. Например, если мы работаем с различными множествами

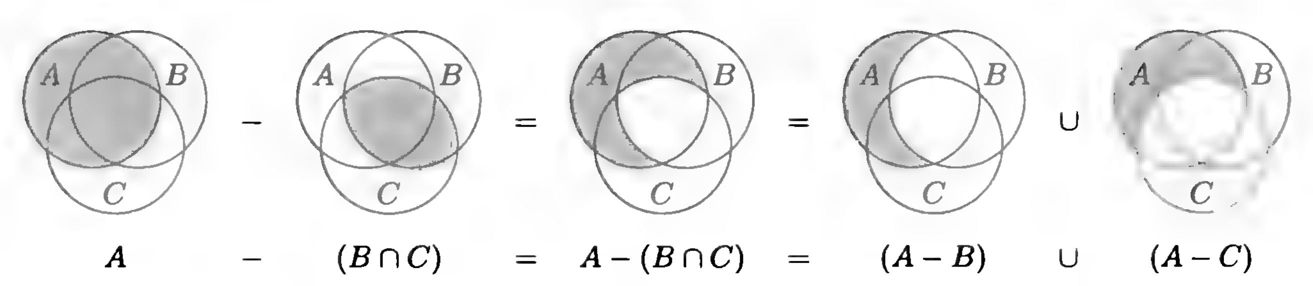


Рис. Б.1. Диаграмма Венна, иллюстрирующая первый закон де Моргана (Б.2). Каждое из множеств A, B и C представлено кругом.

целых чисел, то в качестве универсума можно рассматривать множество $\mathbb{Z}$. Для заданного универсума U можно определить ***дополнение*** (complement) множества A как $\overline{A} = U - A = \{x : x \in U \text{ и } x \notin A\}$. Для любого множества $A \subseteq U$ выполняются следующие соотношения:

$$\overline{\overline{A}} = A \ ,$$
$$A \cap \overline{A} = \emptyset \ ,$$
$$A \cup \overline{A} = U \ .$$

Можно переписать законы де Моргана (Б.2) с дополнениями множеств. Для любых двух множеств $B, C \subseteq U$ имеют место равенства

$$\overline{B \cap C} = \overline{B} \cup \overline{C} \ ,$$
$$\overline{B \cup C} = \overline{B} \cap \overline{C} \ .$$

Два множества, A и B, являются ***непересекающимися*** (disjoint), если они не имеют общих элементов, т.е. если $A \cap B = \emptyset$. Семейство $\mathcal{S} = \{S_i\}$ непустых множеств образует ***разбиение*** (partition) множества S, если

- множества ***попарно не пересекаются***, т.е. из $S_i, S_j \in \mathcal{S}$ и $i \neq j$ следует $S_i \cap S_j = \emptyset$;
- их объединение равно S, т.е.

$$S = \bigcup_{S_i \in \mathcal{S}} S_i \ .$$

Другими словами, $\mathcal{S}$ образует разбиение множества S, если каждый элемент множества S принадлежит ровно одному из множеств $S_i \in \mathcal{S}$.

Количество элементов множества называется его ***мощностью*** (cardinality), или размером, и обозначается как $|S|$. Два множества имеют одинаковую мощность, если между их элементами можно установить взаимно однозначное соответствие. Мощность пустого множества $|\emptyset| = 0$. Если мощность множества представляет собой целое неотрицательное число, то такое множество называется ***конечным***; в противном случае множество является ***бесконечным***. Бесконечное множество, элементы которого могут быть поставлены во взаимно однозначное соответствие натуральным числам $\mathbb{N}$, называются ***счетными*** (countably infinite), в противном случае мы имеем дело с ***несчетными*** (uncountably) множествами. Множество целых чисел $\mathbb{Z}$ счетно, в то время как множество действительных чисел $\mathbb{R}$ — нет.

Для любых двух конечных множеств, A и B, справедливо тождество

$$|A \cup B| = |A| + |B| - |A \cap B| \ , \tag{Б.3}$$

из которого следует неравенство

$$|A \cup B| \leq |A| + |B| \ .$$

Если A и B являются непересекающимися множествами, то $|A \cap B| = 0$, и, таким образом, $|A \cup B| = |A| + |B|$. Если $A \subseteq B$, то $|A| \leq |B|$.

Конечное множество из n элементов иногда называется ***n-элементным***. В англоязычной литературе одноэлементное множество имеет специальное название ***singleton***, а подмножество из k элементов иногда именуется ***k-subset***.

Множество всех подмножеств множества S, включая пустое подмножество и само множество S, обозначается как 2^S и называется ***степенным множеством*** (power set) S. Например, $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$. Мощность степенного множества конечного множества S имеет мощность $2^{|S|}$ (см. упр. Б.1.5).

Иногда мы сталкиваемся с множествоподобными структурами, элементы которых упорядочены. ***Упорядоченная пара*** (ordered pair) состоит из двух элементов, a и b, обозначается как (a,b) и формально может быть определена как множество $(a,b) = \{a, \{a,b\}\}$. Таким образом, упорядоченные пары (a,b) и (b,a) *различны*.

Декартово произведение (cartesian product) двух множеств, A и B, обозначается $A \times B$ и представляет собой множество всех упорядоченных пар, в которых первый элемент пары является элементом A, а второй — элементом B. Говоря более строго,

$$A \times B = \{(a,b) : a \in A \text{ и } b \in B\} \ .$$

Например, $\{a,b\} \times \{a,b,c\} = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c)\}$. Если и A, и B являются конечными множествами, то мощность их декартова произведения равна

$$|A \times B| = |A| \cdot |B| \ . \tag{Б.4}$$

Декартово произведение n множеств $A_1, A_2, \ldots, A_n$ представляет собой множество ***кортежей*** из n элементов (n-tuples)

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i \ , i = 1, 2, \ldots, n\} \ ,$$

мощность которого в случае, если все множества конечны, равна

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n| \ .$$

Мы обозначаем n-кратное декартово произведение единственного множества A на себя как множество

$$A^n = A \times A \times \cdots \times A \ ,$$

мощность которого равна $|A^n| = |A|^n$, если A конечно. Можно также рассматривать кортеж из n элементов как конечную последовательность длиной n (см. с. 1219).

Упражнения

Б.1.1

Изобразите диаграмму Венна, которая иллюстрирует первое из свойств дистрибутивности (Б.1).

Б.1.2

Докажите обобщение законов де Моргана для любого конечного набора множеств:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n}\ ,$$
$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}\ .$$

Б.1.3 ⋆

Докажите обобщение равенства (Б.3), именуемое ***принципом включений и исключений*** (principle of inclusion and exclusion):

$$\begin{aligned} |A_1 \cup A_2 \cup \cdots \cup A_n| = {} & |A_1| + |A_2| + \cdots + |A_n| \\ & - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots && \text{(все пары)} \\ & + |A_1 \cap A_2 \cap A_3| + \cdots && \text{(все тройки)} \\ & \vdots \\ & + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|\ . \end{aligned}$$

Б.1.4

Покажите, что множество нечетных натуральных чисел счетно.

Б.1.5

Покажите, что для любого конечного множества S степенное множество 2^S содержит $2^{|S|}$ элементов (т.е. имеется $2^{|S|}$ различных подмножеств множества S).

Б.1.6

Дайте индуктивное определение кортежа из n элементов, используя понятие упорядоченной пары.

Б.2. Отношения

Бинарным отношением (binary relation) R между элементами двух множеств, A и B, называется подмножество декартова произведения $A \times B$. Если $(a, b) \in R$, это можно записать как $a\,R\,b$. Когда мы говорим, что R есть бинарное отношение на множестве A, имеется в виду, что R является подмножеством $A \times A$. Например, отношение "меньше" на множестве натуральных чисел представляет собой множество $\{(a, b) : a, b \in \mathbb{N}$ и $a < b\}$. Под n-арным отношением на множествах $A_1, A_2, \ldots, A_n$ понимается подмножество декартова произведения $A_1 \times A_2 \times \cdots \times A_n$.

Бинарное отношение $R \subseteq A \times A$ является ***рефлексивным***, если для всех $a \in A$ справедливо

$$a\,R\,a\ .$$

Например, отношения “$=$” и “$\leq$” на множестве $\mathbb{N}$ рефлексивны, но отношение “$<$” таковым не является. Отношение R ***симметрично***, если

$$\text{из} \quad a\,R\,b \quad \text{следует} \quad b\,R\,a$$

для всех $a, b \in A$. Например, симметричным является отношение “$=$”, но не “$<$” или “$\leq$”. Отношение R ***транзитивно***, если

$$\text{из} \quad a\,R\,b \quad \text{и} \quad b\,R\,c \quad \text{следует} \quad a\,R\,c$$

для всех $a, b, c \in A$. Например, транзитивными являются отношения “$=$”, “$<$” и “$\leq$”, но отношение $R = \{(a, b) : a, b \in \mathbb{N} \text{ и } a = b - 1\}$ таковым не является, поскольку из $3\,R\,4$ и $4\,R\,5$ не следует $3\,R\,5$. Отношение, которое является одновременно рефлексивным, симметричным и транзитивным, называется ***отношением эквивалентности***. Например, на множестве натуральных чисел отношение “$=$” является отношением эквивалентности, а отношение “$<$” — нет. Если R — отношение эквивалентности на множестве A, то можно определить ***класс эквивалентности*** элемента $a \in A$ как множество $[a] = \{b \in A : a\ R\ b\}$, т.е. множество всех элементов, эквивалентных a. Например, определим отношение $R = \{(a, b) : a, b \in \mathbb{N} \text{ и } a + b \text{ является четным числом}\}$. В таком случае R является отношением эквивалентности, поскольку $a + a$ четно (рефлексивность), если четно $a + b$, то четно и $b + a$ (симметричность), и из четности $a + b$ и $b + c$ вытекает четность $a + c$ (транзитивность). Класс эквивалентности числа 4 представляет собой $[4] = \{0, 2, 4, 6, \ldots\}$, а класс эквивалентности числа 3 имеет вид $[3] = \{1, 3, 5, 7, \ldots\}$. Основная теорема о классах эквивалентности заключается в следующем.

Теорема Б.1 (Отношения эквивалентности тождественны разбиениям)

Для любого отношения эквивалентности R на множестве A классы эквивалентности образуют разбиение A; и обратно: любое разбиение A определяет отношение эквивалентности на A, для которого множества в разбиении являются классами эквивалентности.

Доказательство. Для доказательства первого утверждения теоремы необходимо показать, что классы эквивалентности R непусты, попарно не пересекаются, и их объединение дает множество A. Из рефлексивности R вытекает $a \in [a]$, так что класс эквивалентности не является пустым; кроме того, поскольку каждый элемент $a \in A$ принадлежит классу эквивалентности $[a]$, объединение всех классов эквивалентности равно A. Остается показать, что классы эквивалентности попарно не пересекаются, т.е. что если два класса эквивалентности, $[a]$ и $[b]$, имеют общий элемент c, то это в действительности один и тот же класс эквивалентности. Предположим, что $a\ R\ c$ и $b\ R\ c$. В силу симметричности $c\ R\ b$, а в силу транзитивности $a\ R\ b$. Таким образом, для произвольного элемента $x \in [a]$ имеем $x\ R\ a$ и, в силу транзитивности, $x\ R\ b$, так что $[a] \subseteq [b]$. Аналогично $[b] \subseteq [a]$, так что $[a] = [b]$.

Для доказательства второй части теоремы рассмотрим разбиение $\mathcal{A} = \{A_i\}$ множества A и определим отношение $R = \{(a, b)$: существует i, такое, что $a \in A_i$ и $b \in A_i\}$ Покажем, что R есть отношение эквивалентности на A. Это отношение рефлексивно, так как из $a \in A_i$ вытекает $a\,R\,a$. Симметричность R следует из того, что если $a\,R\,b$, то и a, и b принадлежат одному и тому же множеству A_i, так что $b\,R\,a$. Если $a\,R\,b$ и $b\,R\,c$, то все три элемента принадлежат одному и тому же множеству A_i, так что $a\,R\,c$, т.е. отношение R транзитивно. Чтобы показать, что множества в разбиении являются классами эквивалентности R, заметим, что если $a \in A_i$, то из $x \in [a]$ вытекает $x \in A_i$, а из $x \in A_i$ вытекает $x \in [a]$. ■

Бинарное отношение R на множестве A является ***антисимметричным***, если

$$\text{из} \quad a\,R\,b \quad \text{и} \quad b\,R\,a \quad \text{следует} \quad a = b\,.$$

Например, антисимметричным на множестве натуральных чисел является отношение "$\leq$", поскольку если $a \leq b$ и $b \leq a$, то $a = b$. Отношение, являющееся одновременно рефлексивным, антисимметричным и транзитивным, является и отношением ***частичного порядка*** (partial order), а множество, на котором определено такое отношение, — ***частично упорядоченным множеством***. Например, отношение "быть потомком" на множестве людей является отношением частичного порядка (если рассматривать человека как собственного потомка).

В частично упорядоченном множестве A может не быть единственного "наибольшего" элемента a, такого, что $b\,R\,a$ для всех $b \in A$. Вместо этого может быть несколько ***максимальных*** элементов a, обладающих тем свойством, что не существует таких $b \in A$, отличных от a, что $a\,R\,b$. Например, в наборе ящиков разного размера может быть несколько максимальных ящиков, которые не могут поместиться ни в один другой ящик, так что одного "наибольшего" ящика, в котором могут поместиться все остальные, не существует[3].

Отношение R является отношением ***полного порядка*** (total order), если для всех $a, b \in A$ имеем $a\,R\,b$ или $b\,R\,a$ (или и то, и другое одновременно), т.е. если любая пара элементов A связана отношением R. Например, отношение "$\leq$" на множестве натуральных чисел является отношением полного порядка, в то время как отношение "быть потомком" на множестве людей таковым не является, поскольку могут быть люди, которые не являются потомками друг друга. Отношение полного порядка, которое является транзитивным, но не обязательно рефлексивным и антисимметричным, называется ***полным прямым порядком*** (total preorder).

---

[3]Для того чтобы отношение "может поместиться в" было отношением частичного порядка, нужно считать, что ящик может поместиться сам в себя.

39 Зак. 3726

Упражнения

Б.2.1

Докажите, что отношение подмножества "$\subseteq$" на множестве всех подмножеств $\mathbb{Z}$ является отношением частичного, но не полного порядка.

Б.2.2

Покажите, что для любого положительного n отношение "тождественности по модулю n" является отношением эквивалентности на множестве целых чисел. (Мы говорим, что $a \equiv b \pmod{n}$, если существует такое целое число q, что $a - b = qn$.) На сколько классов эквивалентности разбивает множество целых чисел это отношение?

Б.2.3

Приведите пример отношений, которые

а. рефлексивны и симметричны, но не транзитивны;

б. рефлексивны и транзитивны, но не симметричны;

в. симметричны и транзитивны, но не рефлексивны.

Б.2.4

Пусть S представляет собой конечное множество, а R является отношением эквивалентности на $S \times S$. Покажите, что если отношение R антисимметрично, то все классы эквивалентности S по отношению к R содержат по одному элементу.

Б.2.5

Профессор полагает, что всякое симметричное и транзитивное отношение R должно быть рефлексивным. Он предлагает следующее доказательство: из $a\,R\,b$ в силу симметрии следует $b\,R\,a$, а применение транзитивности к этому выводу дает $a\,R\,a$. Прав ли профессор?

Б.3. Функции

Для данных двух множеств, A и B, ***функция*** f представляет собой бинарное отношение на A и B, такое, что для всех $a \in A$ существует ровно одно $b \in B$, такое, что $(a, b) \in f$. Множество A называется ***областью определения*** (domain) функции f, а множество B — ***областью значений*** (codomain). Иногда для функции используется запись $f : A \to B$; кроме того, если $(a, b) \in f$, то мы записываем $b = f(a)$, поскольку значение b однозначно определяется значением a.

Интуитивно функцию f можно рассматривать как операцию, которая ставит в соответствие каждому элементу A элемент B. Никакому элементу A не может быть поставлено в соответствие два различных элемента B, однако один и тот же элемент B может соответствовать разным элементам A. Например, бинарное

отношение

$$f = \{(a, b) : a, b \in \mathbb{N} \text{ и } b = a \bmod 2\}$$

является функцией $f : \mathbb{N} \to \{0, 1\}$, поскольку для каждого натурального числа a имеется ровно одно число b из $\{0, 1\}$, такое, что $b = a \bmod 2$. Например, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ и т.д. Бинарное же отношение

$$g = \{(a, b) : a, b \in \mathbb{N} \text{ и } a + b \text{ четно}\}$$

функцией не является, поскольку и $(1, 3)$, и $(1, 5)$ являются элементами g, так что элементу $a = 1$ соответствует более одного элемента b, такого, что $(a, b) \in g$.

Если имеется функция $f : A \to B$ и если $b = f(a)$, то a называется ***аргументом*** функции f, а b — ***значением*** функции f от данного аргумента a. Можно определить функцию, указывая ее значения для каждого элемента из области определения. Например, можно определить $f(n) = 2n$ для $n \in \mathbb{N}$, что означает $f = \{(n, 2n) : n \in \mathbb{N}\}$. Две функции, f и g, называются ***равными***, если у них одинаковые области определения и значений и если для любого a из области определения $f(a) = g(a)$.

Конечная последовательность длиной n представляет собой функцию f, область определения которой — множество из n целых чисел $\{0, 1, \ldots, n - 1\}$. Зачастую конечная последовательность записывается как список ее значений: $\langle f(0), f(1), \ldots, f(n-1)\rangle$. ***Бесконечной последовательностью*** называется функция, область определения которой — множество натуральных чисел $\mathbb{N}$. Например, последовательность чисел Фибоначчи, определенная рекуррентным соотношением (3.22), представляет собой бесконечную последовательность $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Если область определения функции f является декартовым произведением, дополнительные скобки вокруг аргументов в записи обычно опускаются. Например, если есть функция $f : A_1 \times A_2 \times \cdots \times A_n \to B$, то можно записать $b = f(a_1, a_2, \ldots, a_n)$, а не $b = f((a_1, a_2, \ldots, a_n))$. Кроме того, в этом случае ***аргументом*** называется каждый элемент a_i, хотя технически единым аргументом функции f является кортеж из n элементов $(a_1, a_2, \ldots, a_n)$.

Если $b = f(a)$ для некоторой функции $f : A \to B$, то иногда говорят, что b есть ***образ*** (image) a относительно f. Образ подмножества $A' \subseteq A$ относительно f определяется как

$$f(A') = \{b \in B : b = f(a) \text{ для некоторого } a \in A'\} \ .$$

Множество значений (range) функции f представляет собой образ ее области определения, т.е. $f(A)$. Например, множеством значений функции $f : \mathbb{N} \to \mathbb{N}$, определенной как $f(n) = 2n$, является $f(\mathbb{N}) = \{m : m = 2n \text{ для некоторого } n \in \mathbb{N}\}$, другими словами, множество неотрицательных четных целых чисел.

Функция называется ***наложением*** или ***сюръекцией*** (surjection), если ее множество значений совпадает с областью значений. Например, функция $f(n) = \lfloor n/2 \rfloor$ представляет собой наложение $\mathbb{N}$ на $\mathbb{N}$, поскольку каждый элемент этого множества $\mathbb{N}$ служит значением функции f для некоторого аргумента. Функция же

$f(n) = 2n$ не является наложением $\mathbb{N}$ на $\mathbb{N}$, поскольку, например, число 3 не является значением f ни для какого аргумента; однако эта функция является сюръективной функцией от натуральных чисел на четные числа. Сюръекцию $f : A \to B$ иногда называют ***отображением A в B***.

Функция $f : A \to B$ называется ***вложением*** или ***инъекцией*** (injection), если различным аргументам f соответствуют различные значения, т.е. из $a \neq a'$ вытекает $f(a) \neq f(a')$. Например, функция $f(n) = 2n$ является вложением множества $\mathbb{N}$ в $\mathbb{N}$, поскольку каждое четное число b является образом относительно f не более одного элемента из области определения, а именно — $b/2$. Функция $f(n) = \lfloor n/2 \rfloor$ вложением не является, поскольку значение 1, например, получается для двух аргументов — 2 и 3. Инъекции в англоязычной литературе иногда называются функциями однозначного соответствия (one-to-one function).

Функция $f : A \to B$ называется ***биекцией*** (bijection), если она является одновременно инъекцией и сюръекцией. Например, функция $f(n) = (-1)^n \lceil n/2 \rceil$ является биекцией, отображающей множество $\mathbb{N}$ на $\mathbb{Z}$:

$$\begin{aligned} 0 &\to 0 , \\ 1 &\to -1 , \\ 2 &\to 1 , \\ 3 &\to -2 , \\ 4 &\to 2 , \\ &\vdots \end{aligned}$$

Данная функция является инъекцией, поскольку ни один элемент множества целых чисел $\mathbb{Z}$ не является образом более одного аргумента из $\mathbb{N}$. Она также является сюръекцией, поскольку каждый элемент множества $\mathbb{Z}$ является образом некоторого элемента множества $\mathbb{N}$. Следовательно, рассматриваемая функция является биекцией. Биекции называют также ***взаимно однозначными соответствиями*** (one-to-one correspondence), поскольку они делят все элементы областей определения и значений на пары. Биекция множества A в себя само иногда называется ***перестановкой*** множества A.

Если функция f является биекцией, ***обратная*** (inverse) к ней функция f^{-1} определяется следующим образом:

$$f^{-1}(b) = a \text{ тогда и только тогда, когда } f(a) = b .$$

Например, обратной к функции $f(n) = (-1)^n \lceil n/2 \rceil$ является функция

$$f^{-1}(m) = \begin{cases} 2m , & \text{если } m \geq 0 , \\ -2m - 1 , & \text{если } m < 0 . \end{cases}$$

Упражнения

Б.3.1

Пусть A и B — конечные множества, а $f : A \to B$ — некоторая функция. Покажите, что

а. если f является инъекцией, то $|A| \leq |B|$;

б. если f является сюръекцией, то $|A| \geq |B|$.

Б.3.2

Пусть имеется функция $f(x) = x + 1$. Будет ли она биекцией, если ее область определения и область значений — натуральные числа $\mathbb{N}$? А если ее область определения и область значений — целые числа $\mathbb{Z}$?

Б.3.3

Дайте естественное определение обратного к бинарному отношению. (Если отношение является биекцией, то определение должно давать обратную функцию.)

Б.3.4 $\star$

Приведите пример биекции $\mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$.

Б.4. Графы

В этом разделе будут рассмотрены два типа графов — ориентированные и неориентированные. Следует иметь в виду, что терминологию в этой области еще нельзя назвать вполне устоявшейся, так что в литературе можно встретить определения, отличающиеся от приведенных здесь, хотя в основном эти отличия незначительны. Представление графов в памяти компьютера рассматривалось в разделе 22.1.

Ориентированный граф (directed graph) G определяется как пара (V, E), где V — конечное множество, а E — бинарное отношение на V. Множество V называется ***множеством вершин*** (vertex set) графа G, а его элементы — ***вершинами***. Множество E называется ***множеством ребер*** графа G, а его элементы — ***ребрами***. На рис. Б.2, (а) изображен ориентированный граф с множеством вершин $\{1, 2, 3, 4, 5, 6\}$. Вершины на рисунке показаны кружка́ми, а ребра — стрелками. Обратите внимание на возможность существования ***ребер-циклов***, или ***петель*** (self-loops), т.е. ребер, соединяющих вершину с самой собой.

В ***неориентированном графе*** (undirected graph) $G = (V, E)$ множество ребер E состоит из *неупорядоченных* пар вершин, т.е. ребро является множеством $\{u, v\}$, где $u, v \in V$ и $u \neq v$. По соглашению для ребер используется запись (u, v), причем и (u, v), и (v, u) обозначают одно и то же ребро неориентированного графа. В неориентированном графе петли запрещены, так что каждое ребро содержит две разные вершины. На рис. Б.2, (б) показан неориентированный граф с множеством вершин $\{1, 2, 3, 4, 5, 6\}$.

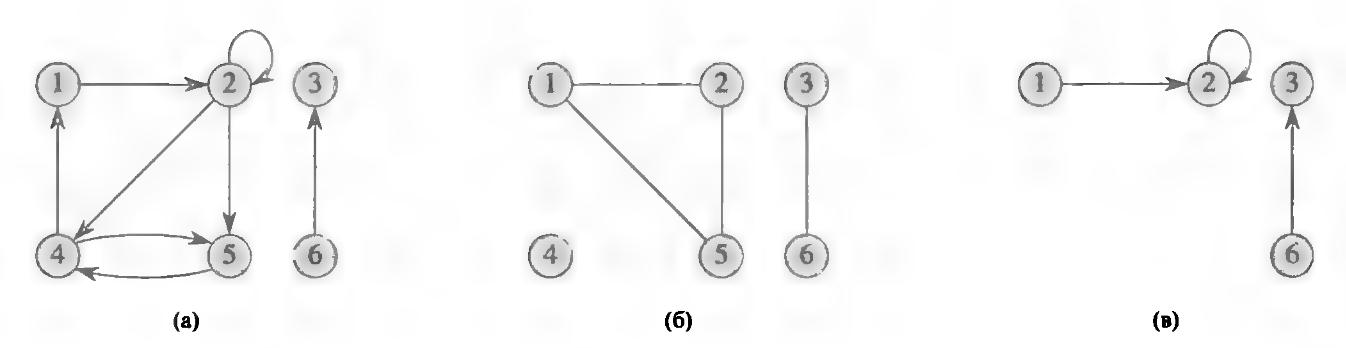


Рис. Б.2. Ориентированные и неориентированные графы. **(а)** Ориентированный граф $G = (V, E)$, где $V = \{1,2,3,4,5,6\}$ и $E = \{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. Ребро $(2,2)$ является петлей. **(б)** Неориентированный граф $G = (V, E)$, где $V = \{1,2,3,4,5,6\}$ и $E = \{(1,2),(1,5),(2,5),(3,6)\}$. Вершина 4 является изолированной. **(в)** Подграф графа из части (а), порожденный множеством вершин $\{1,2,3,6\}$.

Многие определения выглядят одинаково и для ориентированных, и для неориентированных графов, хотя некоторые отличия, естественно, имеются. Если (u, v) — ребро ориентированного графа $G = (V, E)$, то ребро ***выходит из*** (incident from, leave) вершины u и ***входит в*** (incident to, enter) вершину v. Например, на рис. Б.2, (а) из вершины 2 выходят ребра $(2,2)$, $(2,4)$ и $(2,5)$, а входят в вершину 2 ребра $(1,2)$ и $(2,2)$. Если (u, v) — ребро неориентированного графа $G = (V, E)$, то оно ***соединяет вершины*** (incident on) u и v и называется ***инцидентным*** этим вершинам. На рис. Б.2, (б) ребрами, инцидентными вершине 2, являются ребра $(1,2)$ и $(2,5)$.

Если в графе G имеется ребро (u, v), то говорят, что вершина v ***смежна*** (adjacent) с вершиной u. Для неориентированных графов отношение смежности является симметричным (в случае ориентированных графов это утверждение неверно). Если вершина v смежна с вершиной u в ориентированном графе, то иногда пишут $u \to v$. На рис. Б.2 в частях (а) и (б) вершина 2 является смежной с вершиной 1, поскольку ребро $(1,2)$ имеется в обоих графах. Вершина 1 *не* смежна с вершиной 2 на рис. Б.2, (а), поскольку в этом случае ребро $(2,1)$ графу не принадлежит.

Степенью (degree) вершины в неориентированном графе называется число инцидентных ей ребер. Например, вершина 2 на рис. Б.2, (б) имеет степень 2. Вершина, степень которой равна 0 (как, например, у вершины 4 на рис. Б.2, (б)), называется ***изолированной*** (isolated). В ориентированном графе различают ***исходящую степень*** (out-degree), которая равна количеству выходящих из вершины ребер, и ***входящую степень*** (in-degree), которая равна количеству входящих в вершину ребер. ***Степень*** (degree) вершины в ориентированном графе равна сумме ее входящей и исходящей степеней. Так, на рис. Б.2, (а) вершина 2 имеет входящую степень 2, исходящую — 3, так что степень данной вершины равна 5.

Путь (***маршрут***) ***длиной*** k от вершины u к вершине u' в графе $G = (V, E)$ представляет собой последовательность $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ вершин, такую, что $u = v_0$, $u' = v_k$ и $(v_{i-1}, v_i) \in E$ для $i = 1, 2, \ldots, k$. Длиной пути называется количество составляющих его ребер. Путь ***содержит*** (contains) вершины $v_0, v_1, v_2, \ldots, v_k$ и ребра $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Всегда имеется путь нулевой длины из вершины в нее саму. Если имеется путь p из вершины u в вер-

шину u', то говорят, что вершина u' ***достижима*** (reachable) из u по пути p, что иногда в ориентированном графе G записывается как $u \overset{p}{\leadsto} u'$. Путь является ***простым*** (simple), если все вершины пути различны. Например, на рис. Б.2, (а) путь $\langle 1,2,5,4 \rangle$ является простым путем длиной 3; путь $\langle 2,5,4,5 \rangle$ простым не является.

Подпуть (subpath) пути $p = \langle v_0, v_1, \ldots, v_k \rangle$ представляет собой непрерывную подпоследовательность его вершин, т.е. для любых $0 \leq i \leq j \leq k$ последовательность вершин $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ является подпутем пути p.

В ориентированном графе путь $\langle v_0, v_1, \ldots, v_k \rangle$ образует ***цикл*** (cycle), если $v_0 = v_k$ и путь содержит по крайней мере одно ребро. Цикл называется ***простым***, если, кроме того, все вершины $v_1, v_2, \ldots, v_k$ различны. Петля является циклом с длиной 1. Два пути — $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ и $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0 \rangle$ — образуют один и тот же цикл, если существует такое целое j, что $v'_i = v_{(i+j) \bmod k}$ для $i = 0, 1, \ldots, k-1$ (один цикл получен из другого сдвигом). На рис. Б.2, (а) путь $\langle 1,2,4,1 \rangle$ образует тот же цикл, что и пути $\langle 2,4,1,2 \rangle$ и $\langle 4,1,2,4 \rangle$. Этот цикл простой, но цикл $\langle 1,2,4,5,4,1 \rangle$ таковым не является. Цикл $\langle 2,2 \rangle$, образованный ребром $(2,2)$, представляет собой петлю. Ориентированный граф, не содержащий петель, называется ***простым***. В неориентированном графе путь $\langle v_0, v_1, \ldots, v_k \rangle$ образует ***цикл***, если $k > 0$, $v_0 = v_k$ и все ребра этого пути различны; цикл является ***простым***, если вершины $v_1, v_2, \ldots, v_k$ различны. Например, на рис. Б.2, (б) путь $\langle 1,2,5,1 \rangle$ представляет собой простой цикл. Граф, в котором нет простых циклов, называется ***ациклическим***.

Неориентированный граф является ***связным*** (connected), если каждая его вершина достижима из всех прочих вершин. ***Связные компоненты*** неориентированного графа представляют собой классы эквивалентности вершин относительно достижимости (отношения "быть достижимым из"). Граф на рис. Б.2, (б) имеет три связных компонента: $\{1,2,5\}$, $\{3,6\}$ и $\{4\}$. Каждая вершина в $\{1,2,5\}$ достижима из другой вершины этого множества. Неориентированный граф является связным тогда и только тогда, когда он состоит из единственного связного компонента. Ребра связного компонента инцидентны только вершинам этого компонента; другими словами, ребро (u,v) является ребром связного компонента, только если и u, и v являются вершинами этого компонента.

Ориентированный граф называется ***сильно связным*** (strongly connected), если любые его две вершины достижимы друг из друга. Любой ориентированный граф можно разбить на ***сильно связные компоненты*** (strongly connected components), которые определяются как классы эквивалентности отношения "являются взаимно достижимыми". Ориентированный граф считается сильно связным тогда и только тогда, когда он состоит из единственного сильно связного компонента. Граф на рис. Б.2, (а) состоит из трех таких компонентов: $\{1,2,4,5\}$, $\{3\}$ и $\{6\}$. Все пары в множестве $\{1,2,4,5\}$ являются взаимно достижимыми. Вершины $\{3,6\}$ не образуют сильно связный компонент, поскольку из вершины 3 нельзя достичь вершины 6.

Два графа, $G = (V,E)$ и $G' = (V',E')$, ***изоморфны*** (isomorphic), если существует биекция $f : V \to V'$, такая, что $(u,v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E'$. Другими словами, мы можем перенумеровать вершины G, превратив их в вершины G', но сохранив при этом ребра между вершина-

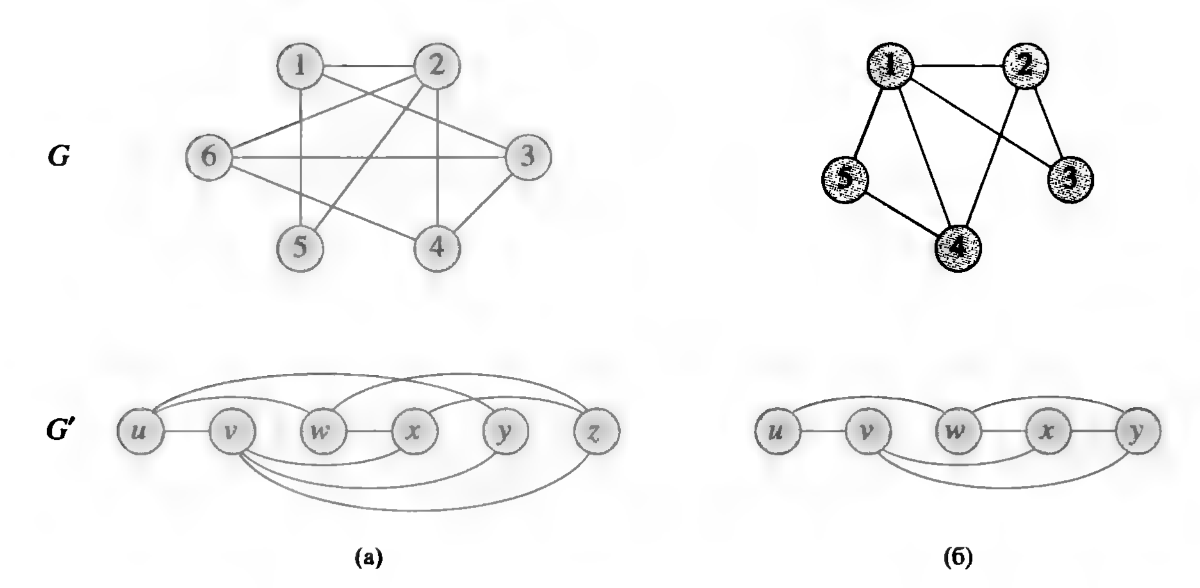


Рис. Б.3. **(а)** Пара изоморфных графов. Вершины верхнего графа отображаются на вершины нижнего графа следующим образом: $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. **(б)** Два графа, не являющихся изоморфными, поскольку верхний граф имеет вершину степени 4, которой нет у нижнего графа.

ми в неизменном состоянии. На рис. Б.3,(а) показана пара изоморфных графов G и G' с множествами вершин $V = \{1,2,3,4,5,6\}$ и $V' = \{u,v,w,x,y,z\}$ соответственно. Отображение V на V' выглядит следующим образом: $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. Графы на рис. Б.3,(б) неизоморфны. Хотя оба изображенных здесь графа имеют по 5 вершин и 7 ребер, граф в верхней части рисунка имеет вершину степени 4, которой нет у нижнего графа.

Мы говорим, что граф $G' = (V', E')$ является ***подграфом*** (subgraph) графа $G = (V, E)$, если $V' \subseteq V$ и $E' \subseteq E$. Если в графе $G = (V, E)$ выбрано подмножество $V' \subseteq V$, то подграфом графа G, ***порожденным*** (induced) множеством вершин V', является граф $G' = (V', E')$, где

$$E' = \{(u,v) \in E : u,v \in V'\} \ .$$

Подграф, порожденный множеством вершин $\{1,2,3,6\}$ на рис. Б.2,(а), показан на рис. Б.2,(в), и содержит следующее множество ребер: $\{(1,2),(2,2),(6,3)\}$.

Для данного неориентированного графа $G = (V, E)$ его ***ориентированная версия*** (directed version) представляет собой ориентированный граф $G' = (V, E')$, где $(u,v) \in E'$ тогда и только тогда, когда $(u,v) \in E$. Другими словами, каждое неориентированное ребро (u,v) графа G заменяется в ориентированной версии двумя ориентированными ребрами — (u,v) и (v,u). Для ориентированного графа $G = (V, E)$ его ***неориентированная версия*** (undirected version) представляет собой неориентированный граф $G' = (V, E')$, где $(u,v) \in E'$ тогда и только тогда, когда $u \neq v$ и $(u,v) \in E$. Другими словами, неориентированная версия содержит ребра графа G "с удаленными стрелочками", причем петли из неориентированной версии убираются. Поскольку в неориентированном графе ребра (u,v) и (v,u) идентичны, неориентированная версия ориентированного графа содержит ребро только по разу, даже если в ориентированном графе между вершинами u и v имеются два ориентированных ребра — (u,v) и (v,u). В ориентированном графе

$G = (V, E)$ ***соседом*** (neighbor) вершины u называется любая вершина, которая становится смежной с u в неориентированной версии, т.е. v является соседом u, если $u \neq v$ и либо $(u, v) \in E$, либо $(v, u) \in E$. В неориентированном графе вершины являются соседями, если они смежные.

Некоторые виды графов имеют свои специальные названия. ***Полным*** (complete) графом называется неориентированный граф, в котором каждая пара вершин образована смежными вершинами, т.е. который содержит все возможные ребра. ***Двудольным*** (bipartite) называется неориентированный граф $G = (V, E)$, в котором множество V может быть разделено на два множества, V_1 и V_2, такие, что из $(u, v) \in E$ следует, что либо $u \in V_1$ и $v \in V_2$, либо $u \in V_2$ и $v \in V_1$. Ациклический неориентированный граф называется ***лесом*** (forest), а связный ациклический неориентированный граф — (***свободным***) ***деревом*** (free tree) (см. раздел Б.5).

Имеется еще два варианта графов, с которыми вы можете встретиться. Это ***мультиграф*** (multigraph), который похож на неориентированный граф, но может содержать как петли, так и по нескольку ребер между вершинами. ***Гиперграф*** (hypergraph) также похож на неориентированный граф, но содержит ***гиперребра***, которые могут соединять произвольное количество вершин. Многие алгоритмы, разработанные для обычных ориентированных и неориентированных графов, могут быть обобщены для работы с такими графоподобными структурами.

Сжатием (contraction) неориентированного графа $G = (V, E)$ по ребру $e = (u, v)$ называется граф $G' = (V', E')$, где $V' = V - \{u, v\} \cup \{x\}$ (x — новая вершина). Множество ребер E' образуется из E путем удаления ребра (u, v), и, кроме того, для каждой вершины w, инцидентной к u или v, удаляются ребра (u, w) и (v, w) (если они имеются в E) и добавляется новое ребро — (x, w). По сути, u и v "сжимаются" в одну вершину.

Упражнения

Б.4.1

На приеме каждый гость подсчитывает, сколько рукопожатий он сделал. По окончании приема вычисляется сумма рукопожатий каждого из гостей. Покажите, что полученная сумма четна, доказав следующую ***лемму о рукопожатиях***: если $G = (V, E)$ — неориентированный граф, то сумма степеней всех вершин равна удвоенному числу ребер

$$\sum_{v \in V} \text{degree}(v) = 2\,|E| \ .$$

Б.4.2

Покажите, что если ориентированный или неориентированный граф содержит путь из вершины u в вершину v, то в нем есть простой путь между u и v. Покажите, что если в ориентированном графе есть цикл, то в нем есть простой цикл.

Б.4.3

Покажите, что для любого связного неориентированного графа $G = (V, E)$ выполняется соотношение $|E| \geq |V| - 1$.

Б.4.4

Покажите, что отношение "быть достижимым из" в неориентированном графе является отношением эквивалентности на множестве вершин графа. Какие из трех свойств отношения эквивалентности выполняются для отношения "быть достижимым из" в ориентированном графе?

Б.4.5

Изобразите неориентированную версию графа, показанного на рис. Б.2, (а), и ориентированную версию графа, показанного на рис. Б.2, (б).

Б.4.6 ★

Покажите, что гиперграф можно представить как двудольный граф, в котором отношение смежности соответствует отношению инцидентности в гиперграфе. (*Указание:* одно множество вершин двудольного графа должно соответствовать вершинам гиперграфа, а другое множество — гиперребрам.)

Б.5. Деревья

Как и слово "граф", слово "дерево" употребляется в нескольких родственных смыслах. Здесь представлены основные определения и математические свойства некоторых видов деревьев. Вопросы представления деревьев в памяти компьютера рассматриваются в разделах 10.4 и 22.1.

Б.5.1. Свободные деревья

Как уже говорилось в разделе Б.4, ***свободные деревья*** (free tree), или ***деревья без выделенного корня***, представляют собой связный ациклический неориентированный граф. Прилагательное "свободный" зачастую опускается, когда речь идет о графе, являющемся деревом. Если неориентированный граф ацикличный, но, возможно, несвязный, то он является ***лесом***. Многие алгоритмы, разработанные для деревьев, могут работать и с лесом. Пример свободного дерева показан на рис. Б.4, (а), а леса — на рис. Б.4, (б). Лес на рис. Б.4, (б) не является деревом в си-

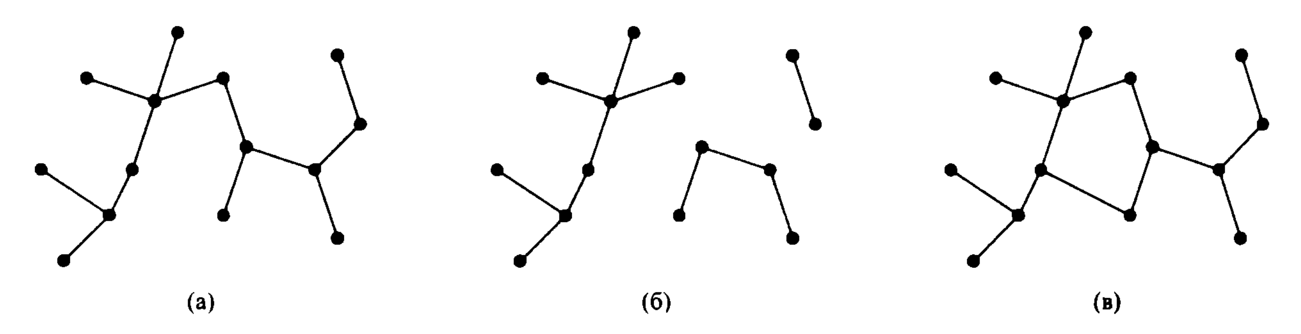


Рис. Б.4. **(а)** Свободное дерево. **(б)** Лес. **(в)** Граф, содержащий цикл, а потому не являющийся ни деревом, ни лесом.

лу того, что представляющий его граф не является связным. Граф на рис. Б.4, (в) связный, но содержит цикл, а потому не может быть ни деревом, ни лесом.

В следующей теореме указано несколько важных свойств деревьев.

Теорема Б.2 (Свойства свободных деревьев)

Пусть $G = (V, E)$ является неориентированным графом. Тогда следующие утверждения равносильны.

1. G — свободное дерево.
2. Любые две вершины G соединяются с помощью единственного простого пути.
3. G — связный граф, но при удалении из E любого ребра граф перестает быть таковым.
4. G — связный граф, и $|E| = |V| - 1$.
5. G — ациклический граф, и $|E| = |V| - 1$.
6. G — ациклический граф, но при добавлении любого ребра в E получается граф, содержащий цикл.

Доказательство. (1) $\Rightarrow$ (2): поскольку дерево является связным, для любых двух вершин G имеется как минимум один соединяющий их простой путь. Предположим для доказательства от противного, что u и v — вершины, соединенные двумя простыми путями, p_1 и p_2, как показано на рис. Б.5. Пусть w — вершина, в которой пути впервые расходятся, т.е. следующие за w вершины на путях p_1 и p_2 соответственно x и y, причем $x \neq y$. Пусть z — первая вершина, в которой пути вновь сходятся, т.е. z — первая после w вершина на пути p_1, которая также принадлежит пути p_2. Пусть p' — подпуть p_1 из w через x в z, а p'' — подпуть p_2 из w через y в z. Пути p' и p'' не имеют общих точек, кроме конечных. Тогда путь, образованный объединением p' и пути, обратного к p'', образует цикл. Это противоречит нашему предположению о том, что G является деревом. Таким образом, если G — дерево, то между вершинами не может быть больше одного соединяющего их простого пути.

(2) $\Rightarrow$ (3): если любые две вершины G соединяются единственным простым путем, то G — связный граф. Пусть (u, v) — ребро из E. Это ребро представляет

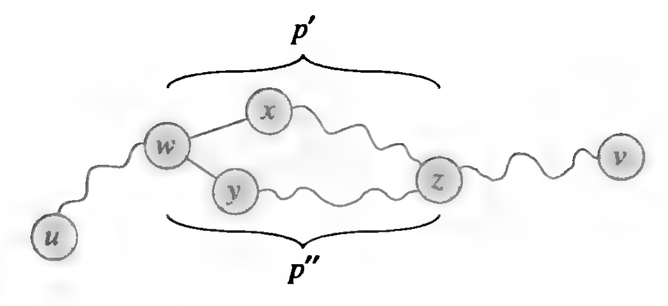


Рис. Б.5. Шаг доказательства теоремы Б.2: если (1) G является свободным деревом, то (2) две любые вершины в G связаны единственным простым путем. Предполагаем для доказательства от противного, что вершины u и v связаны двумя различными простыми путями p_1 и p_2. Эти пути впервые расходятся в вершине w, и впервые вновь сходятся в вершине z. Путь p' при соединении с путем p'' образует цикл, что приводит к противоречию.

собой путь из u в v, а значит, это единственный путь из u в v. Если мы удалим из графа путь (u, v), пути из u в v не будет, а граф перестанет быть связным.

(3) $\Rightarrow$ (4): по условию граф G является связным, а из упр. Б.4.3 известно, что $|E| \geq |V| - 1$. Докажем по индукции, что $|E| \leq |V| - 1$. Связный граф с одной или двумя вершинами имеет ребер на одно меньше, чем вершин. Предположим, что G имеет $n \geq 3$ вершин и что все графы, удовлетворяющие (3) с менее чем n вершинами удовлетворяют также условию $|E| \leq |V| - 1$. Удаление произвольного ребра из G разделяет граф на $k \geq 2$ связных компонентов (на самом деле $k = 2$). Каждый компонент удовлетворяет условию (3) теоремы, так как в противном случае этому условию не удовлетворяет само дерево G. Если рассматривать каждый компонент V_i с множеством ребер E_i как отдельное свободное дерево, то, поскольку каждый компонент имеет меньше чем $|V|$ вершин, по индукции мы имеем $|E_i| \leq |V_i| - 1$. Таким образом, количество ребер во всех компонентах не превышает $|V| - k \leq |V| - 2$. Добавление удаленного ребра приводит к $|E| \leq |V| - 1$.

(4) $\Rightarrow$ (5): предположим, что граф G является связным и что $|E| = |V| - 1$. Необходимо показать, что граф G ациклический. Предположим, что граф содержит цикл, состоящий из k вершин $v_1, v_2, \ldots, v_k$; без потери общности можно считать, что это простой цикл. Пусть $G_k = (V_k, E_k)$ — подграф G, состоящий из данного цикла. Заметим, что $|V_k| = |E_k| = k$. Если $k < |V|$, то должна существовать вершина $v_{k+1} \in V - V_k$, смежная с некоторой вершиной $v_i \in V_k$, что следует из связности G. Определим подграф $G_{k+1} = (V_{k+1}, E_{k+1})$ графа G как подграф, у которого $V_{k+1} = V_k \cup \{v_{k+1}\}$ и $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Заметим, что $|V_{k+1}| = |E_{k+1}| = k+1$. Если $k+1 < |V|$, мы можем продолжить построение, аналогично определяя G_{k+2}, и так до тех пор, пока не получим $G_n = (V_n, E_n)$, такой, что $n = |V|$, $V_n = V$ и $|E_n| = |V_n| = |V|$. Поскольку G_n — подграф G, $E_n \subseteq E$, следовательно, $|E| \geq |V|$, что противоречит предположению $|E| = |V| - 1$. Следовательно, граф G ациклический.

(5) $\Rightarrow$ (6): предположим, что граф G ациклический и что $|E| = |V| - 1$. Пусть k — количество связных компонентов графа G. По определению каждый связный компонент представляет собой свободное дерево; поскольку из (1) вытекает (5), сумма всех ребер во всех связных компонентах G равна $|V| - k$. Следовательно, k должно быть равно 1, а G должно быть деревом. Поскольку из (1) вытекает (2), любые две вершины G соединены единственным простым путем. Таким образом, добавление любого ребра к G создает цикл.

(6) $\Rightarrow$ (1): предположим, что G — ациклический граф, но добавление любого ребра в E приводит к образованию цикла. Мы должны показать, что G — связный граф. Пусть u и v — произвольные вершины графа G. Если u и v не являются смежными, добавление ребра (u, v) создает цикл, в котором все ребра, кроме (u, v), принадлежат G. Таким образом, существует путь из u в v, но поскольку u и v выбраны произвольно, G оказывается связным графом. ■

Б.5.2. Корневые и упорядоченные деревья

Корневое дерево (rooted tree) представляет собой свободное дерево, в котором выделена одна вершина, именуемая ***корнем*** (root) дерева. Зачастую вершины в дереве с корнем называют ***узлами***[4] (nodes) дерева. На рис. Б.6, (а) показано дерево с корнем, состоящее из 12 узлов с корнем в узле 7.

Рассмотрим узел x в дереве T с корнем r. Любой узел y на единственном простом пути от r к x называется ***предком*** (ancestor) x. Если y является предком x, то x является ***потомком*** (descendant) y (каждый узел является собственным предком и потомком). Если y — предок x и $x \neq y$, то y — ***истинный предок*** (proper ancestor) x, а x, соответственно, ***истинный потомок*** (proper descendant) y. ***Поддеревом с корнем в узле x*** (subtree rooted at x) называется дерево, порожденное потомками x, корнем которого является узел x. Например, на рис. Б.6, (а) поддерево с корнем в узле 8 содержит узлы 8, 6, 5 и 9.

Если (y, x) — последнее ребро на пути из корня r дерева T в узел x, то узел y является ***родительским*** (parent) по отношению к x, а x — ***ребенком*** (child), или ***дочерним*** узлом по отношению к узлу y. Корень дерева — единственный узел, не имеющий родительского узла. Если два узла имеют общий родительский узел, мы будем называть такие узлы ***родственными***, или ***братьями*** (siblings). Узел, у которого нет дочерних узлов, называется ***внешним узлом*** (external node) или ***листом*** (leaf). Узел, не являющийся листом, называется ***внутренним узлом*** (internal node).

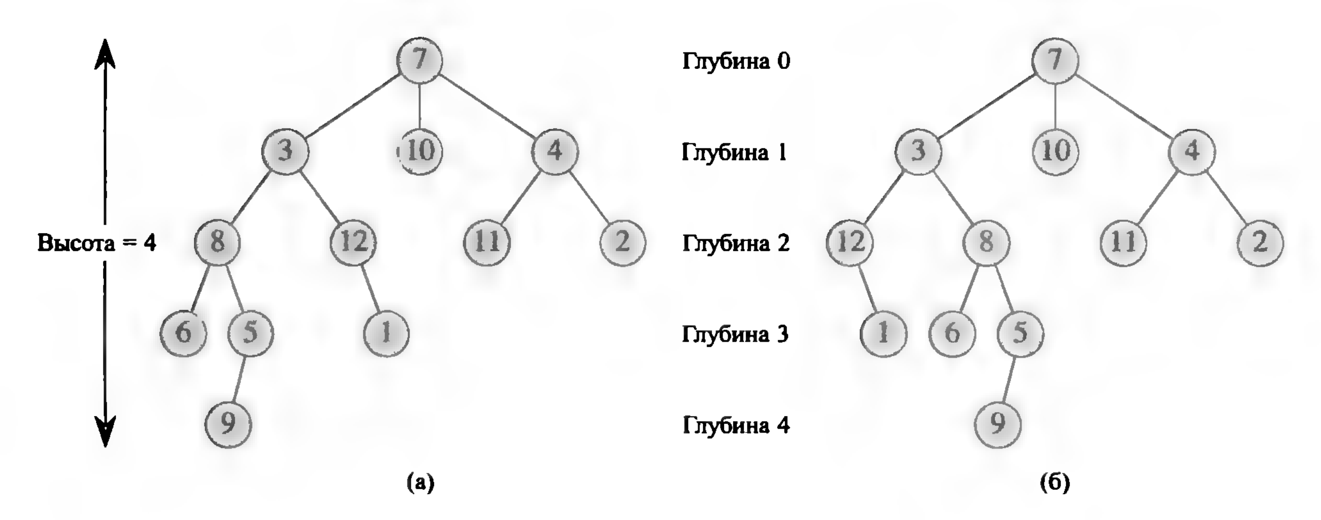


Рис. Б.6. Корневые и упорядоченные деревья. **(а)** Корневое дерево высотой 4. Дерево изображено стандартным способом: корень (узел 7) — вверху, его дочерние узлы (узлы на глубине 1) расположены под ним, их дочерние узлы (узлы на глубине 2) расположены под ними, и т.д. Если дерево упорядочено, имеет значение относительный порядок слева направо дочерних узлов; в противном случае дерево не упорядочено. **(б)** Другое упорядоченное дерево. В качестве корневого дерева оно идентично дереву из части (а), но в качестве упорядоченного дерева отличается, поскольку дочерние узлы узла 3 располагаются в другом порядке.

---

[4]Термин "узел" зачастую используется в теории графов как синоним термина "вершина". Мы же будем использовать термин "узел" только для вершины дерева с корнем.

Количество дочерних узлов узла x называется его ***степенью***[5] (degree). Длина простого пути из корня r в узел x называется ***глубиной*** (depth) узла x в дереве. ***Высота*** (height) узла в дереве равна количеству ребер в самом длинном простом нисходящем пути от узла к листу; высотой дерева называется высота его корня. Высота дерева также равна наибольшей глубине узла дерева.

Упорядоченное дерево (ordered tree) представляет собой дерево с корнем, в котором дочерние узлы каждого узла упорядочены, т.е. если узел имеет k дочерних узлов, то у него имеется первый, второй, $\ldots$, k-й дочерние узлы. Два дерева, приведенные на рис. Б.6, отличаются, если рассматривать их как упорядоченные деревья, но одинаковы, если трактовать их как обычные корневые деревья.

Б.5.3. Бинарные и позиционные деревья

Бинарные деревья определяются рекурсивно. ***Бинарное дерево*** (binary tree) T представляет собой конечное множество узлов, которое

- либо не содержит узлов;
- либо состоит из трех непересекающихся множеств узлов: ***корневой*** узел, бинарное дерево, называемое ***левым поддеревом*** (left subtree), и бинарное дерево, называемое ***правым поддеревом*** (right subtree).

Бинарное дерево, которое не содержит узлов, называется ***пустым*** (empty tree) или ***нулевым*** (null tree) и иногда обозначается как NIL. Если левое поддерево непустое, его корень называется ***левым ребенком*** (left child) корня всего дерева; аналогично корень непустого правого поддерева называется ***правым ребенком*** (right child). Если поддерево является пустым, мы говорим, что соответствующий ребенок ***отсутствует*** (absent, missing). Пример бинарного дерева можно увидеть на рис. Б.7, (а).

Бинарное дерево представляет собой не просто упорядоченное дерево, в котором каждый узел имеет степень не более 2. Например, в бинарном дереве в случае узла с одним дочерним узлом имеет значение, какой именно этот дочерний узел — левый или правый. В упорядоченных деревьях такое различие в случае одного дочернего узла не делается. На рис. Б.7, (б) показано бинарное дерево, отличающееся от приведенного на рис. Б.7, (а), позицией одного узла. Если рассматривать эти деревья как просто упорядоченные, то они являются идентичными.

Можно представить позиционную информацию в бинарном дереве с помощью внутренних узлов упорядоченного дерева, как показано на рис. Б.7, (в). Идея заключается в том, чтобы заменить каждый отсутствующий дочерний узел узлом, не имеющим потомков. Эти листья на рисунке изображены квадратами. Так получается ***полностью бинарное дерево*** (full binary tree): каждый его узел либо

---

[5] Заметим, что степень узла зависит от того, рассматриваем ли мы дерево с корнем или свободное дерево. Степень вершины в свободном дереве, как и в любом неориентированном графе, равна количеству смежных вершин. В дереве с корнем степень равна количеству дочерних узлов — родительский узел при вычислении степени не учитывается.

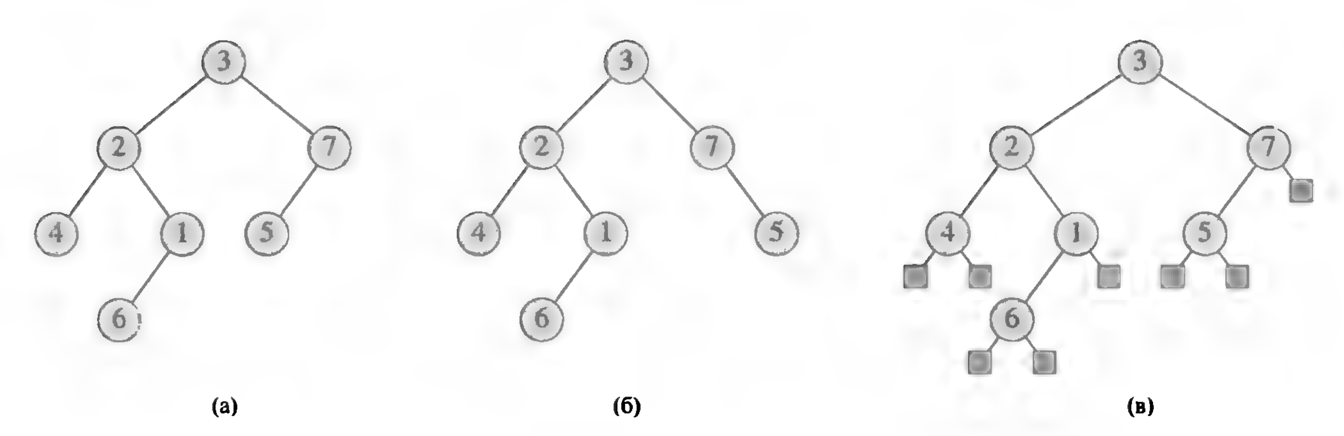


Рис. Б.7. Бинарные деревья. **(а)** Бинарное дерево, изображенное в стандартном виде. Левый ребенок узла изображается ниже и левее этого узла; правый ребенок изображается ниже и правее. **(б)** Бинарное дерево, отличное от дерева из части (а). В части (а) левым дочерним узлом узла 7 является узел 5, а правый дочерний узел отсутствует. В части (б) у узла 7 отсутствует левый дочерний узел, а узел 5 является его правым дочерним узлом. Как упорядоченные деревья деревья в частях (а) и (б) идентичны, но как бинарные деревья они различны. **(в)** Бинарное дерево из (а), представленное внутренними узлами полностью бинарного дерева (упорядоченного дерева, в котором каждый внутренний узел имеет степень 2). Листья этого дерева показаны квадратиками.

представляет собой лист, либо имеет степень 2. Узлы со степенью 1 в таком дереве отсутствуют. Следовательно, порядок дочерних узлов сохраняет информацию о местоположении.

Информация о позициях узлов, которая отличает упорядоченные деревья от бинарных, может быть расширена на случай деревьев более чем с 2 дочерними узлами в каждом узле. В ***позиционном дереве*** (positional tree) все дочерние узлы данного узла пронумерованы различными натуральными числами. Если у данного узла среди дочерних нет узла с номером i, то i-й дочерний узел у данного узла ***отсутствует*** (absent). Позиционное дерево называется ***k-арным*** (k-ary) деревом, если в нем нет дочерних узлов с номером, превышающим k. Таким образом, бинарное дерево представляет собой k-арное дерево при $k = 2$.

Полным k-арным деревом (complete k-ary tree) называется k-арное дерево, все листья которого имеют одну и ту же глубину, а все внутренние узлы — одну и ту же степень k. Так, на рис. Б.8 показано полное бинарное дерево, высота которого равна 3. Сколько листьев содержится в полном k-арном дереве, высота которого равна h? Корень имеет k дочерних узлов на глубине 1, каждый из которых содержит по k дочерних узлов с глубиной 2 и т.д. Таким образом, количество листьев на глубине h равно k^h. Соответственно, высота полного k-арного дерева с n листьями равна $\log_k n$. Количество внутренних узлов полного k-арного дерева высоты h равно

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

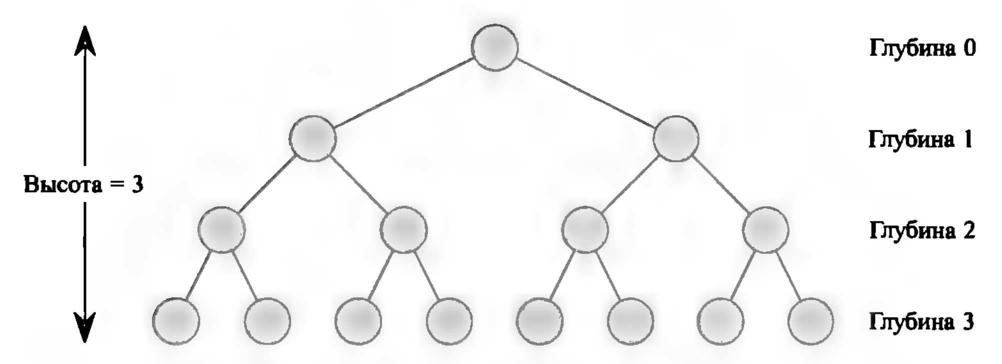


Рис. Б.8. Полное бинарное дерево высотой 3 с восемью листьями и семью внутренними узлами.

согласно (A.5). Таким образом, полное бинарное дерево содержит $2^h - 1$ внутренних узлов.

Упражнения

Б.5.1

Нарисуйте все свободные деревья, состоящие из трех вершин — x, y и z. Изобразите все корневые деревья с этими же узлами и узлом x в качестве корня. Нарисуйте все упорядоченные деревья с узлами x, y и z и узлом x в качестве корня. Изобразите все бинарные деревья с этими же узлами и узлом x в качестве корня.

Б.5.2

Пусть $G = (V, E)$ — ориентированный ациклический граф, в котором имеется вершина $v_0 \in V$, такая, что имеется единственный путь из v_0 в каждую другую вершину $v \in V$. Докажите, что неориентированная версия графа G является деревом.

Б.5.3

Покажите по индукции, что количество узлов степени 2 в любом непустом бинарном дереве на единицу меньше количества листьев. Сделайте отсюда вывод о том, что количество внутренних узлов полного бинарного дерева на единицу меньше количества листьев.

Б.5.4

Покажите с использованием метода математической индукции, что непустое бинарное дерево с n узлами имеет высоту как минимум $\lfloor \lg n \rfloor$.

Б.5.5 ★

Определим ***длину внутреннего пути*** (internal path length) полностью бинарного дерева как сумму глубин всех внутренних узлов дерева. Аналогично под ***длиной внешнего пути*** (external path length) будем подразумевать сумму глубин всех листьев дерева. Рассмотрим полностью бинарное дерево с n внутренними узлами, длиной внутреннего пути i и длиной внешнего пути e. Докажите, что $e = i + 2n$.

Б.5.6 ★

Назначим каждому листу x с глубиной d бинарного дерева T "вес" $w(x) = 2^{-d}$. Докажите, что $\sum_{x \in L} w(x) \leq 1$ (***неравенство Крафта*** (Kraft inequality)).

Б.5.7 ★

Покажите, что если $L \geq 2$, то каждое бинарное дерево с L листьями содержит поддерево с количеством листьев от $L/3$ до $2L/3$ включительно.

Задачи

Б.1. Раскраска графа

Определим для заданного неориентированного графа $G = (V, E)$ ***k-раскраску*** (k-coloring) графа G как функцию $c : V \to \{0, 1, \ldots, k-1\}$, такую, что $c(u) \neq c(v)$ для каждого ребра $(u, v) \in E$. Другими словами, числа $0, 1, \ldots, k-1$ представляют k цветов, и смежные вершины должны иметь разные цвета.

а. Покажите, что любое дерево можно раскрасить двумя цветами.

б. Покажите, что следующие утверждения эквивалентны:

1. граф G двудольный;
2. граф G раскрашивается двумя цветами;
3. граф G не имеет циклов нечетной длины.

в. Пусть d — максимальная степень вершины в графе G. Докажите, что граф G может быть раскрашен $d+1$ цветами.

г. Покажите, что если граф имеет $O(|V|)$ ребер, то его можно раскрасить $O(\sqrt{|V|})$ цветами.

Б.2. Граф дружбы

Преобразуйте следующие утверждения в теоремы о неориентированных графах и докажите их. Отношение дружбы считаем симметричным, но не рефлексивным.

а. В любой группе из $n \geq 2$ человек имеется два человека с одним и тем же количеством друзей из этой группы.

б. Каждая группа из шести человек содержит либо три человека, которые являются друзьями друг друга, либо три человека, никакие два из которых не являются друзьями.

в. Любую группу людей можно разделить на две подгруппы так, что как минимум половина друзей каждого человека из одной подгруппы будет находиться в *другой* подгруппе.

г. Если каждый человек в группе является другом по меньшей мере для половины группы, то можно рассадить эту группу людей за столом так, что каждый будет сидеть между двумя друзьями.

Б.3. Разбиение деревьев

Многие алгоритмы типа "разделяй и властвуй", работающие с графами, требуют разбиения графа на два близких по размеру подграфа, порождаемых разбиением множества вершин. Вопрос заключается в том, как сделать это с наименьшим количеством удаляемых ребер. Должно выполняться условие, что если две вершины находятся в конечном итоге в одном и том же поддереве после удаления ребер, то они должны находиться и в одном и том же разбиении вершин.

а. Покажите, что удалением единственного ребра можно разбить вершины любого бинарного дерева с n вершинами на два множества, A и B, такие, что $|A| \leq 3n/4$ и $|B| \leq 3n/4$.

б. Покажите, что константу $3/4$ из п. (а) нельзя улучшить. Для этого приведите пример простого бинарного дерева, для которого при удалении любого ребра в одной из частей оказывается ровно $3n/4$ вершин.

в. Покажите, что, удаляя не более $O(\lg n)$ ребер, можно разбить бинарное дерево с n вершинами на такие два множества, A и B, что $|A| = \lfloor n/2 \rfloor$ и $|B| = \lceil n/2 \rceil$.

Заключительные замечания

Основатель символьной логики Дж. Буль (G. Boole) ввел многие обозначения, связанные с множествами, в своей книге, изданной в 1854 году. Современная теория множеств (в первую очередь, теория мощности бесконечных множеств) была создана Г. Кантором (G. Cantor) в 1874–1895 годах. Термин "функция" введен Г.В. Лейбницем (G.W. Leibniz) для некоторых типов математических формул. Его весьма ограниченное определение функции позже неоднократно обобщалось и расширялось. Создание теории графов относится к 1736 году, когда Л. Эйлер (L. Euler) доказал невозможность такого обхода семи мостов в Кенигсберге, при котором выполняется по одному проходу по каждому из мостов и обход завершается в исходной точке.

Полезным справочником, содержащим множество определений и свойств графов, является книга Харари (Harary) [159].

Приложение В. Комбинаторика и теория вероятности

В этом приложении вы познакомитесь с азами комбинаторики и теории вероятности. Если вы уже знакомы с этими разделами математики, то можете просто бегло ознакомиться с началом приложения и обратить больше внимания на его окончание. В большинстве глав этой книги теория вероятности не используется, но некоторые целиком построены на ее применении.

В разделе В.1 приведен обзор основ комбинаторики, включая формулы для количества перестановок и сочетаний. В разделе В.2 вас ожидает встреча с аксиомами теории вероятности и основами распределения вероятностей. Случайные величины, а также математическое ожидание и дисперсия рассматриваются в разделе В.3. Раздел В.4 посвящен геометрическому и биномиальному распределениям, изучение которых продолжается в разделе В.5, где обсуждается проблема "хвостов" распределений.

В.1. Основы комбннаторики

Комбинаторика пытается ответить на вопрос "Сколько?", не выполняя перечисления. Например, вы можете спросить "Сколько всего имеется различных n-битовых чисел?" или "Сколькими способами можно упорядочить n различных чисел?" Здесь вы познакомитесь с азами комбинаторики, которые предполагают знание основ теории множеств, так что, надеемся, вы основательно проработали раздел Б.1.

Правила суммы и произведения

Иногда множество, количество элементов которого необходимо подсчитать, можно выразить как объединение непересекающихся множеств или как декартово произведение множеств.

Правило суммы гласит, что количество способов, которыми можно выбрать элемент из одного из двух *непересекающихся* множеств, равно сумме мощностей этих множеств. То есть, если A и B — два конечных множества без общих членов, то $|A \cup B| = |A| + |B|$, что следует из уравнения (Б.3). Например, если каждый символ в номере машины должен быть либо латинской буквой, либо цифрой, то всего имеется $26 + 10 = 36$ различных вариантов выбора этого символа, так как всего имеется 26 вариантов выбора буквы и 10 — цифры.

Правило произведения гласит, что количество способов, которыми можно выбрать упорядоченную пару, равно количеству вариантов выбора первого элемента, умноженному на количество вариантов выбора второго элемента. То есть, если A и B — конечные множества, то $|A \times B| = |A| \cdot |B|$ (см. уравнение (Б.4)). Например, имея 28 сортов мороженого и 4 разных сиропа, можно приготовить $28 \cdot 4 = 112$ различных вариантов мороженого с сиропом.

Строки

Строкой (string) на конечном множестве S называют последовательность элементов S. Например, вот восемь двоичных (составленных из 0 и 1) строк длиной 3:

$$000, 001, 010, 011, 100, 101, 110, 111 \ .$$

Иногда строки длиной k называются ***k-строками***. ***Подстрокой*** (substring) s' строки s называется упорядоченная последовательность элементов s. Подстрока длиной k называется ***k-подстрокой***. Например, 010 — 3-подстрока строки 01101001 (начинающаяся с четвертой позиции строки); 111 же подстрокой указанной строки не является.

k-строка на множестве S может рассматриваться как элемент декартова произведения k-кортежей S^k, так что всего имеется $|S|^k$ строк длиной k, в частности — число двоичных k-строк равно 2^k. Интуитивно это очевидно: при построении k-строки из множества с n элементами имеется n вариантов выбора первого элемента строки; для каждого из первых элементов имеется n вариантов выбора второго элемента строки, и так k раз. Это приводит нас к k-кратному произведению и дает общее количество k-строк, равное $n \cdot n \cdots n = n^k$.

Перестановки

Перестановкой (permutation) конечного множества S называется упорядоченная последовательность всех элементов S, в которой каждый элемент встречается ровно один раз. Например, если $S = \{a, b, c\}$, то имеется шесть перестановок S:

$$abc, acb, bac, bca, cab, cba \ .$$

Всего имеется $n!$ перестановок множества из n элементов, поскольку первый элемент может быть выбран n способами, второй — $n - 1$ способом, третий — $n - 2$ и т.д.

k-перестановкой[1] S называется упорядоченная последовательность k элементов из S, в которой ни один элемент не встречается дважды (таким образом, обычная перестановка представляет собой n-перестановку множества из n элементов. Для множества $\{a, b, c, d\}$ имеется двенадцать 2-перестановок:

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc \ .$$

---

[1] В отечественной литературе k-перестановка называется ***размещением***. — *Примеч. пер.*

Количество k-перестановок множества из n элементов равно

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}\,, \tag{В.1}$$

поскольку имеется n способов выбора первого элемента, $n-1$ — второго и так до последнего, k-го элемента, который можно выбрать из оставшихся $n-k+1$ элементов множества.

Сочетания

Сочетаниями (k-combination) из n элементов по k называются k-элементные подмножества n-элементного множества. Например, имеется шесть сочетаний по два элемента из множества $\{a, b, c, d\}$:

$$ab, ac, ad, bc, bd, cd\ .$$

(Здесь для простоты для записи подмножества $\{a, b\}$ использована краткая запись ab.) Для построения сочетания из множества просто выбирается k различных элементов. Порядок выбора элементов значения не имеет.

Количество сочетаний можно выразить через количество размещений. Для каждого сочетания имеется $k!$ перестановок его элементов, каждая из которых представляет собой одно из размещений из n элементов по k. Таким образом, количество сочетаний из n элементов по k равно количеству размещений, деленному на $k!$, т.е. с учетом (В.1) количество сочетаний из n элементов по k равно

$$\frac{n!}{k!\,(n-k)!}\ . \tag{В.2}$$

При $k = 0$ эта формула дает единицу, т.е. выбрать пустое подмножество можно единственным способом (напомним, что $0! = 1$).

Биномиальные коэффициенты

Для числа сочетаний из n элементов по k используется обозначение $\binom{n}{k}$[2]. Из (В.2) следует, что

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}\ .$$

Эта формула симметрична относительно k и $n-k$:

$$\binom{n}{k} = \binom{n}{n-k}\ . \tag{В.3}$$

[2] В отечественной литературе для этой величины принято обозначение C_n^k — *Примеч. пер.*

Эти числа известны также как ***биномиальные коэффициенты***, поскольку они участвуют в ***биноме Ньютона***:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} . \tag{B.4}$$

В частном случае $x = y = 1$ мы получаем

$$2^n = \sum_{k=0}^{n} \binom{n}{k} .$$

Комбинаторный смысл этой формулы заключается в подсчете количества двоичных строк длиной n (число которых равно 2^n) как суммы количества строк с разным числом единиц (имеется $\binom{n}{k}$ двоичных строк длиной n с k единицами, так как $\binom{n}{k}$ — количество способов выбрать k позиций для единиц в строке длиной n).

Имеется масса различных тождеств, в которых принимают участие биномиальные коэффициенты (с некоторыми из них вы познакомитесь в упражнениях к данному разделу).

Оценки биномиальных коэффициентов

В некоторых случаях нам может потребоваться оценить величину биномиальных коэффициентов и указать их границы. Нижняя граница для $1 \leq k \leq n$ может быть оценена следующим образом:

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\geq \left(\frac{n}{k}\right)^k \end{aligned}$$

Используя неравенство $k! \geq (k/e)^k$, являющееся следствием из формулы Стирлинга (3.18), можно получить оценку верхней границы биномиальных коэффициентов:

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k . \end{aligned} \tag{B.5}$$

Для всех целых $0 \leq k \leq n$ по индукции можно доказать (см. упр. В.1.12), что

$$\binom{n}{k} \leq \frac{n^n}{k^k(n-k)^{n-k}} , \tag{B.6}$$

где для удобства принято, что $0^0 = 1$. Для $k = \lambda n$, где $0 \leq \lambda \leq 1$, это неравенство можно переписать как

$$\begin{aligned} \binom{n}{\lambda n} &\leq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n \\ &= 2^{n\,H(\lambda)} , \end{aligned}$$

где

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{B.7}$$

называется (***двоичной***) ***энтропийной функцией*** (binary entropy function). Для удобства принято, что $0 \lg 0 = 0$, так что $H(0) = H(1) = 0$.

Упражнения

В.1.1

Сколько k-подстрок имеется у n-строки? (Одинаковые подстроки, начинающиеся в разных позициях строки, считаются разными.) Сколько всего подстрок имеется у строки длиной n?

В.1.2

Булева функция (boolean function) с n входами и m выходами — это функция с областью определения $\{\text{TRUE}, \text{FALSE}\}^n$ и областью значений $\{\text{TRUE}, \text{FALSE}\}^m$. Сколько всего имеется различных функций с n входами и одним выходом? С n входами и m выходами?

В.1.3

Сколькими способами n профессоров могут разместиться на конференции за круглым столом? Варианты, отличающиеся поворотом, считаются одинаковыми.

В.1.4

Сколькими способами можно выбрать из множества $\{1, 2, \ldots, 99\}$ три различных числа так, чтобы их сумма была четной?

В.1.5

Докажите тождество

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{B.8}$$

для $0 < k \leq n$.

В.1.6

Докажите тождество

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

для $0 \le k < n$.

В.1.7

При выборе k объектов из n один из объектов можно пометить специальным образом и отслеживать, выбран он или нет. Используя этот подход, докажите, что

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} .$$

В.1.8

Используя результат упр. В.1.7, составьте таблицу биномиальных коэффициентов $\binom{n}{k}$ для $n = 0, 1, \ldots, 6$ и $0 \le k \le n$ в виде равнобедренного треугольника (в первой строке — $\binom{0}{0}$, во второй — $\binom{1}{0}$ и $\binom{1}{1}$ и т.д.). Такая таблица биномиальных коэффициентов называется ***треугольником Паскаля***.

В.1.9

Докажите, что

$$\sum_{i=1}^{n} i = \binom{n+1}{2} .$$

В.1.10

Покажите, что для любых целых чисел $n \ge 0$ и $0 \le k \le n$ выражение $\binom{n}{k}$ достигает максимального значения при $k = \lfloor n/2 \rfloor$ или $k = \lceil n/2 \rceil$.

В.1.11 ★

Докажите, что для любых целых чисел $n \ge 0$, $j \ge 0$, $k \ge 0$ и $j + k \le n$

$$\binom{n}{j+k} \le \binom{n}{j}\binom{n-j}{k} . \qquad \text{(B.9)}$$

Приведите как алгебраическое доказательство данного неравенства, так и доказательство, основанное на рассуждениях о выборе $j + k$ элементов из n. Приведите пример, когда выполняется строгое неравенство.

В.1.12 ★

Докажите неравенство (В.6) по индукции по всем $0 \le k \le n/2$, а затем воспользуйтесь уравнением (В.3) для распространения результата на все $0 \le k \le n$.

В.1.13 ★

Воспользуйтесь приближением Стирлинга для доказательства того, что

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)) . \qquad \text{(B.10)}$$

В.1.14 ★

Дифференцируя энтропийную функцию $H(\lambda)$, покажите, что ее максимум достигается при $\lambda = 1/2$. Чему равно значение $H(1/2)$?

В.1.15 ★

Покажите, что для любого целого $n \geq 0$

$$\sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1} . \qquad \text{(В.11)}$$

В.2. Вероятность

Вероятность является очень важным инструментом в процессе разработки и анализа вероятностных и рандомизированных алгоритмов. В этом разделе вы познакомитесь с основами теории вероятности.

Мы определим вероятность с помощью ***пространства выборок*** (sample space) S, которое представляет собой множество ***элементарных событий*** (elementary events). Каждое элементарное событие может рассматриваться как возможный исход некоторого эксперимента. Например, в случае эксперимента, состоящего в подбрасывании двух различимых монеток пространство выборок состоит из всех возможных 2-строк над множеством $\{\text{О}, \text{Р}\}$ (где О обозначает выпадение орла, а Р — решки):

$$S = \{\text{ОО}, \text{ОР}, \text{РО}, \text{РР}\} .$$

Событие (event) представляет собой подмножество[3] пространства выборок S. Например, в эксперименте с бросанием двух монет событием может быть выпадение одного орла и одной решки: $\{\text{ОР}, \text{РО}\}$. Событие S называется ***достоверным событием*** (certain event), а событие $\emptyset$ — ***невозможным*** (null event). Мы говорим, что два события, A и B, являются взаимоисключающими (mutually exclusive), если $A \cap B = \emptyset$. Каждое элементарное событие $s \in S$ также будет рассматриваться нами как событие $\{s\}$. Все элементарные события по определению являются взаимоисключающими.

---

[3]В случае обобщенного распределения вероятностей могут иметься некоторые подмножества пространства выборок S, которые не рассматриваются как события. Такая ситуация обычно возникает, когда пространство выборок несчетно бесконечное. Главное требование, которому должны отвечать подмножества, чтобы быть событиями, заключается в том, что множество событий пространства выборок должно быть замкнуто относительно операций дополнения к событию, объединения конечного или счетного числа событий и пересечения конечного или счетного числа событий. Большинство распределений вероятностей, с которыми нам придется иметь дело, — над конечным или счетным пространством выборок, так что мы в общем случае будем рассматривать все подмножества пространства выборок как события. Важным исключением является непрерывное равномерное распределение вероятностей, с которым мы вскоре встретимся.

Аксиомы вероятности

Распределение вероятностей (probability distribution) $\Pr\{\}$ на пространстве выборок S отображает события из S на действительные числа, удовлетворяя при этом ***аксиомам вероятности***.

1. $\Pr\{A\} \geq 0$ для любого события A.
2. $\Pr\{S\} = 1$.
3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ для любых двух взаимоисключающих событий A и B. В общем случае для любой (конечной или счетной бесконечной) последовательности попарно взаимоисключающих событий $A_1, A_2, \ldots$

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\} \ .$$

Мы называем $\Pr\{A\}$ ***вероятностью*** (probability) события A. Заметим, что аксиома 2 выполняет нормализующее действие: нет никаких фундаментальных оснований в выборе в качестве вероятности достоверного события именно единицы; просто такое значение наиболее естественное и удобное.

Некоторые результаты следуют непосредственно из приведенных аксиом и основ теории множеств (см. раздел Б.1). Невозможное событие $\emptyset$ имеет вероятность $\Pr\{\emptyset\} = 0$. Если $A \subseteq B$, то $\Pr\{A\} \leq \Pr\{B\}$. Используя запись $\overline{A}$ для обозначения события $S - A$ (***дополнения*** (complement) A), получим $\Pr\{\overline{A}\} = 1 - \Pr\{A\}$. Для любых двух событий A и B

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \tag{B.12}$$
$$\leq \Pr\{A\} + \Pr\{B\} \ . \tag{B.13}$$

Предположим, что в нашем примере с бросанием монет вероятность каждого из четырех элементарных событий равна $1/4$. Тогда вероятность получения как минимум одного орла равна

$$\begin{aligned}\Pr\{\text{оо}, \text{ор}, \text{ро}\} &= \Pr\{\text{оо}\} + \Pr\{\text{ор}\} + \Pr\{\text{ро}\} \\ &= 3/4 \ .\end{aligned}$$

Другой способ получить эту вероятность — это заметить, что единственный способ получить при бросании меньше одного орла — это выпадение двух решек, вероятность чего равна $\Pr\{\text{рр}\} = 1/4$, так что вероятность получить по крайней мере одного орла равна $1 - 1/4 = 3/4$.

Дискретные распределения вероятностей

Распределение вероятностей называется ***дискретным*** (discrete), если оно определено на конечном или бесконечном счетном пространстве выборок. Пусть

S — пространство выборок. Тогда для любого события A

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\} \ ,$$

поскольку элементарные события, составляющие A, являются взаимоисключающими. Если S конечно и каждое элементарное событие $s \in S$ имеет вероятность

$$\Pr\{s\} = 1/|S| \ ,$$

то мы имеем дело с ***равномерным распределением вероятностей*** (uniform probability distribution) на S. В таком случае эксперимент часто описывается словами "выберем случайным образом элемент S".

В качестве примера рассмотрим бросание ***симметричной монеты*** (fair coin), для которой вероятность выпадения орла равна вероятности выпадения решки и составляет $1/2$. Если мы бросаем монету n раз, то получаем равномерное распределение вероятностей на пространстве выборок $S = \{\text{О}, \text{Р}\}^n$ (которое представляет собой множество размером 2^n). Каждое элементарное событие S может быть представлено строкой длиной n на множестве $\{\text{О}, \text{Р}\}$, и вероятность появления каждой строки равна $1/2^n$. Событие

$$A = \{\text{выпало ровно } k \text{ орлов и } n - k \text{ решек}\}$$

представляет собой подмножество S размером $|A| = \binom{n}{k}$, поскольку имеется ровно $\binom{n}{k}$ строк длиной n на множестве $\{\text{О}, \text{Р}\}$, содержащих ровно k О. Вероятность события A, таким образом, равна $\Pr\{A\} = \binom{n}{k}/2^n$.

Непрерывное равномерное распределение вероятности

Непрерывное равномерное распределение вероятности представляет собой пример распределения вероятности, в котором не все подмножества пространства событий рассматриваются как события. Непрерывное равномерное распределение вероятности определено на закрытом отрезке $[a, b]$ действительных чисел ($a < b$). Интуитивно все точки отрезка $[a, b]$ рассматриваются как "равновероятные". Имеется несчетное множество точек, так что мы не можем назначить каждой точке свою конечную положительную вероятность, так как мы не сможем одновременно удовлетворить аксиомы 2 и 3. По этой причине мы должны связывать вероятность только с *некоторыми* из подмножеств S, чтобы удовлетворить аксиомы вероятности для этих событий.

Непрерывное равномерное распределение вероятностей представляет собой пример распределения вероятностей, в котором не все подмножества пространства выборок рассматриваются как события. Непрерывное равномерное распределение вероятностей определено на закрытом отрезке $[a, b]$ действительных чисел, где $a < b$. Интуитивно оно заключается в том, что каждая точка отрезка $[a, b]$ "равновероятна". Однако в отрезке имеется несчетное количество точек, так что, назначив каждой точке одну и ту же конечную положительную вероятность, мы не сможем одновременно удовлетворить аксиомам 2 и 3. По этой причине мы

связываем вероятность только с *некоторыми* из подмножеств S таким образом, чтобы для этих событий удовлетворялись все аксиомы.

Для любого закрытого отрезка $[c, d]$, где $a \leq c \leq d \leq b$, ***непрерывное равномерное распределение вероятностей*** (continuous uniform probability distribution) определяет вероятность события $[c, d]$ как

$$\Pr\{[c, d]\} = \frac{d - c}{b - a} .$$

Обратите внимание, что для произвольной отдельной точки $x = [x, x]$ вероятность x равна 0. Если мы удалим конечные точки отрезка $[c, d]$, то получим открытый интервал (c, d). Поскольку $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, аксиома 3 дает $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Вообще говоря, множество событий для непрерывного равномерного распределения вероятностей представляет собой произвольное подмножество пространства выборок $[a, b]$, которое может быть получено конечным (или бесконечным счетным) объединением открытых и закрытых интервалов.

Условная вероятность и независимость

Иногда мы располагаем частичной информацией о результате эксперимента. Например, пусть известно, что в результате бросания двух симметричных монет по крайней мере на одной из них выпал орел. Чему в таком случае равна вероятность того, что обе монеты выпали орлом? Имеющаяся информация позволяет исключить выпадение двух решек, а три оставшихся элементарных события имеют равную вероятность $1/3$, так что именно такой и будет интересующая нас вероятность выпадения двух орлов.

Изложенная идея предварительных знаний об эксперименте формализуется в определении ***условной вероятности*** (conditional probability) события A при условии осуществления события B:

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{B.14}$$

(при этом $\Pr\{B\} \neq 0$). ("$\Pr\{A \mid B\}$" читается как "вероятность A при условии B".) Интуитивно формула легко объяснима. Если произошло событие B, то событие, заключающееся в том, что при этом произошло и A, — $A \cap B$, т.е. $A \cap B$ представляет собой множество исходов, когда произошли и A, и B. Поскольку такой исход является одним из элементарных событий B, нормализуем вероятности всех элементарных событий в B, деля их на $\Pr\{B\}$, чтобы их сумма стала равна единице. Следовательно, условная вероятность события A при условии осуществления события B представляет собой отношение вероятности события $A \cap B$ к вероятности события B. В приведенном выше примере A представляет собой событие, заключающееся в выпадении двух орлов, а B — событие, заключающееся в том, что выпал по крайней мере один орел. Итак, $\Pr\{A \mid B\} = (1/4)/(3/4) = 1/3$.

Два события называются ***независимыми*** (independent), если

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} \ , \tag{B.15}$$

что при $\Pr\{B\} \neq 0$ эквивалентно условию

$$\Pr\{A \mid B\} = \Pr\{A\} \ .$$

Например, предположим, что при бросании двух монет результаты отдельных бросков независимы. Тогда вероятность выпадения двух орлов равна $(1/2)(1/2) = 1/4$. Предположим теперь, что одно событие состоит в том, что первая монета выпала орлом, а второе в том, что монеты выпали по-разному. Каждое из этих событий имеет вероятность $1/2$, а вероятность осуществления обоих — $1/4$. В соответствии с определением эти события независимы, несмотря на то что, на первый взгляд, это не очевидно. И наконец представим, что монеты спаяны вместе, так что они либо обе выпадают орлами, либо обе выпадают решками (вероятности этих выпадений равны). Итак, вероятность выпадения каждой монеты орлом — $1/2$, но и вероятность того, что обе монеты выпадут орлом, — также $1/2$, а поскольку $1/2 \neq (1/2)(1/2)$, события "первая монета выпала орлом" и "вторая монета выпала орлом" в данном случае не являются независимыми.

События $A_1, A_2, \ldots, A_n$ называются ***попарно независимыми*** (pairwise independent), если для всех $1 \leq i < j \leq n$ выполняется равенство

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} \ .$$

Мы говорим, что эти события ***независимы в совокупности*** (mutually independent), если для любого k-подмножества $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ исходного множества, где $2 \leq k \leq n$ и $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, выполняется равенство

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\} \ .$$

Например, предположим, что мы бросили две симметричные монеты. Пусть A_1 — событие, заключающееся в выпадении орлом первой монеты, A_2 — событие, заключающееся в выпадении орлом второй монеты, а A_3 — что монеты выпали поразному. Мы имеем

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 \ , \\
\Pr\{A_2\} &= 1/2 \ , \\
\Pr\{A_3\} &= 1/2 \ , \\
\Pr\{A_1 \cap A_2\} &= 1/4 \ , \\
\Pr\{A_1 \cap A_3\} &= 1/4 \ , \\
\Pr\{A_2 \cap A_3\} &= 1/4 \ , \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0 \ .
\end{aligned}$$

Поскольку для $1 \leq i < j \leq 3$ имеем $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, события A_1, A_2 и A_3 попарно независимы, однако не являются независимыми

в совокупности, поскольку $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ и $\Pr\{A_1\} \Pr\{A_2\} \Pr\{A_3\} = 1/8 \neq 0$.

Теорема Байеса

Из определения условной вероятности (В.14) и коммутативности $A \cap B = B \cap A$ следует, что для событий A и B с ненулевыми вероятностями

$$\begin{aligned}\Pr\{A \cap B\} &= \Pr\{B\} \Pr\{A \mid B\} \\ &= \Pr\{A\} \Pr\{B \mid A\} .\end{aligned} \tag{B.16}$$

Решая уравнение относительно $\Pr\{A \mid B\}$, получаем формулу

$$\Pr\{A \mid B\} = \frac{\Pr\{A\} \Pr\{B \mid A\}}{\Pr\{B\}} , \tag{B.17}$$

известную как ***теорема Байеса*** (Bayes's theorem). Знаменатель $\Pr\{B\}$ представляет собой нормирующую константу, которую можно записать иначе. Поскольку $B = (B \cap A) \cup (B \cap \overline{A})$ и поскольку $B \cap A$ и $B \cap \overline{A}$ являются взаимоисключающими событиями,

$$\begin{aligned}\Pr\{B\} &= \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} \\ &= \Pr\{A\} \Pr\{B \mid A\} + \Pr\{\overline{A}\} \Pr\{B \mid \overline{A}\} .\end{aligned}$$

Подставив полученное выражение в (В.17), получим эквивалентный вид формулы Байеса:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\} \Pr\{B \mid A\}}{\Pr\{A\} \Pr\{B \mid A\} + \Pr\{\overline{A}\} \Pr\{B \mid \overline{A}\}} . \tag{B.18}$$

Теорема Байеса может упростить вычисление условной вероятности. Предположим, например, что есть симметричная монета и монета, которая всегда выпадает орлом. Наш эксперимент состоит из трех событий: мы случайным образом выбираем монету и два раза ее подбрасываем. Предположим, что оба раза выпал орел. Какова вероятность того, что мы выбрали несимметричную монету?

Эта задача легко решается с помощью формулы Байеса. Пусть A — событие, состоящее в том, что выбрана несимметричная монета, а B — выпадение двух орлов. Необходимо найти вероятность $\Pr\{A \mid B\}$. Мы имеем $\Pr\{A\} = 1/2$, $\Pr\{B \mid A\} = 1$, $\Pr\{\overline{A}\} = 1/2$ и $\Pr\{B \mid \overline{A}\} = 1/4$; следовательно,

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} \\ &= 4/5 .\end{aligned}$$

Упражнения

В.2.1

Профессор Ванстоун бросает симметричную монету один раз, а профессор Унопетри — два раза. Чему равна вероятность того, что профессор Ванстоун получит больше выпавших орлов, чем профессор Унопетри?

В.2.2

Докажите ***неравенство Буля***: для любой конечной или бесконечной счетной последовательности событий $A_1, A_2, \ldots$ справедливо следующее неравенство:

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \cdots \ . \tag{В.19}$$

В.2.3

Имеется тщательно перетасованная колода из десяти карт, пронумерованных числами от 1 до 10. Из колоды последовательно вынимаются три карты. Какова вероятность того, что эти карты будут находиться в порядке возрастания их достоинства?

В.2.4

Докажите, что

$$\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} = 1 \ .$$

В.2.5

Докажите, что для любого набора событий $A_1, A_2, \ldots, A_n$

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}\} \ .$$

В.2.6 ★

Разработайте процедуру, которая получает в качестве входных параметров два целых числа, a и b ($0 < a < b$), и, используя симметричную монету, имитирует бросание несимметричной монеты с появлением орла с вероятностью a/b и решки — с вероятностью $(b - a)/b$. Оцените математическое ожидание количества бросков монеты (которое должно быть $O(1)$). (*Указание:* используйте бинарное представление числа a/b.)

В.2.7 ★

Покажите, как построить множество из n попарно независимых событий так, чтобы никакое его подмножество из более чем двух элементов не являлось независимым в совокупности.

В.2.8 ★

Два события, A и B, являются ***условно независимыми*** относительно события C, если

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\} \ .$$

Приведите нетривиальный пример двух событий, которые не являются независимыми, но при этом условно независимы относительно некоторого третьего события.

B.2.9 ⋆

Вы участвуете в игровом шоу, в котором приз спрятан за одним из трех занавесов. Вы выигрываете приз, если угадываете, где он находится. После того как вы выбираете занавес, ведущий открывает один из оставшихся. Если за ним приза нет, вы можете изменить свой выбор, указав на третий занавес. Как изменятся ваши шансы на выигрыш, если вы сделаете это?

B.2.10 ⋆

Один из трех заключенных, X, Y и Z, будет освобожден, а остальные будут отбывать наказание. Кто именно — знает охранник, но ему запрещено сообщать заключенным об их судьбе. Заключенный X просит охранника сообщить, кто из Y и Z останется в тюрьме, мотивируя это тем, что он и так знает, что один из них будет отбывать наказание, так что из ответа охранника он ничего не узнает о собственной судьбе. Охранник сообщает, что Y освобожден не будет. После этого X полагает, что его шансы на свободу, которые составляли $1/3$, возрастают до $1/2$, так как он знает, что освобожден будет только либо он, либо Z. Прав ли он или его шансы на свободу остаются равными $1/3$? Обоснуйте свой ответ.

B.3. Дискретные случайные величины

Дискретная случайная величина (discrete random variable) X — это функция, отображающая конечное или бесконечное счетное пространство выборок S на множество действительных чисел. Она связывает с каждым возможным исходом эксперимента действительное число, что позволяет работать с распределением вероятностей на множестве значений данной функции. Случайные величины могут быть определены и для несчетных бесконечных пространств выборок, но при этом возникают технические проблемы, которых лучше избежать. Нам не придется работать с несчетными бесконечными пространствами выборок, так что далее везде предполагается, что случайные величины дискретны.

Для случайной величины X и действительного числа x определим событие $X = x$ как $\{s \in S : X(s) = x\}$; таким образом,

$$\Pr\{X = x\} = \sum_{s \in S : X(s) = x} \Pr\{s\} .$$

Функция

$$f(x) = \Pr\{X = x\}$$

является ***функцией плотности вероятности*** (probability density function) случайной величины X. Из аксиом вероятности следует, что $\Pr\{X = x\} \geq 0$ и $\sum_x \Pr\{X = x\} = 1$.

В качестве примера рассмотрим эксперимент, состоящий в бросании пары игральных костей. В пространстве выборок имеется 36 элементарных событий. Будем считать, что все они равновероятны, т.е. для каждого элементарного события $s \in S$ его вероятность $\Pr\{s\} = 1/36$. Определим случайную величину X как *максимальное* количество очков, выпавшее при броске на одной из костей. Тогда, например, $\Pr\{X = 3\} = 5/36$, так как X равно 3 в 5 из 36 возможных элементарных событий, а именно — при выпадении $(1,3)$, $(2,3)$, $(3,3)$, $(3,2)$ и $(3,1)$.

Зачастую на одном пространстве выборок определяется несколько случайных величин. Если X и Y — случайные величины, то функция

$$f(x, y) = \Pr\{X = x \text{ и } Y = y\}$$

называется ***совместной функцией плотности вероятности*** (joint probability density function) X и Y. Для фиксированного значения y

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ и } Y = y\} ,$$

и аналогично для фиксированного значения x

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ и } Y = y\} .$$

Используя определение условной вероятности (В.14), имеем

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ и } Y = y\}}{\Pr\{Y = y\}} .$$

Определим две случайные величины, X и Y, как ***независимые***, если для всех x и y события $X = x$ и $Y = y$ независимы, или, что то же самое, если для всех x и y мы имеем $\Pr\{X = x \text{ и } Y = y\} = \Pr\{X = x\} \Pr\{Y = y\}$.

Для заданного множества случайных величин, определенных на одном и том же пространстве выборок, можно определить новую случайную величину как сумму, произведение или другую функцию имеющихся случайных величин.

Математическое ожидание случайной величины

Простейшая и наиболее часто используемая характеристика случайной величины — ее "среднее" значение. ***Ожидаемое значение*** (или ***математическое ожидание*** (expected value), или ***среднее*** (mean)) дискретной случайной величины X представляет собой

$$\mathrm{E}[X] = \sum_x x \cdot \Pr\{X = x\} , \qquad \text{(В.20)}$$

которое является вполне определенным, если сумма конечна или абсолютно сходящаяся[4]. Иногда математическое ожидание X обозначают как μ_X или, если случайная величина очевидна из контекста, просто как μ.

Рассмотрим игру, в которой бросается пара симметричных монет. Вы выигрываете три доллара при выпадении орла и проигрываете два доллара при выпадении решки. Математическое ожидание случайной величины X, равной вашему выигрышу, равно

$$\begin{aligned} \mathrm{E}[X] &= 6 \cdot \Pr\{2\ \text{о}\} + 1 \cdot \Pr\{1\ \text{о},\ 1\ \text{р}\} - 4 \cdot \Pr\{2\ \text{р}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1\ . \end{aligned}$$

Математическое ожидание суммы двух случайных величин равно сумме их математических ожиданий:

$$\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]\ , \tag{B.21}$$

если значения $\mathrm{E}[X]$ и $\mathrm{E}[Y]$ определены. Это свойство называется ***линейностью математического ожидания*** (linearity of expectation), причем оно справедливо даже тогда, когда случайные величины X и Y не являются независимыми. Это правило распространяется и на конечные, и на абсолютно сходящиеся бесконечные суммы математических ожиданий. Линейность математического ожидания является ключевым свойством, обеспечивающим возможность проведения вероятностного анализа с использованием индикаторных случайных величин (см. раздел 5.2).

Если X — произвольная случайная величина, то любая функция $g(x)$ определяет новую случайную величину $g(X)$. Если математическое ожидание этой величины определено, то

$$\mathrm{E}[g(X)] = \sum_x g(x) \cdot \Pr\{X = x\}\ .$$

Пусть $g(x) = ax$. Тогда для любой константы a

$$\mathrm{E}[aX] = a\mathrm{E}[X]\ . \tag{B.22}$$

Следовательно, математическое ожидание представляет собой линейную функцию: для любых двух произвольных случайных величин X и Y и произвольной константы a

$$\mathrm{E}[aX+Y] = a\mathrm{E}[X] + \mathrm{E}[Y]\ . \tag{B.23}$$

---

[4]В отечественной математической литературе для математического ожидания принято обозначение $\mathrm{M}[X]$. — *Примеч. ред.*

Если случайные величины X и Y независимы и каждая имеет определенное математическое ожидание, то

$$\begin{aligned} \mathrm{E}[XY] &= \sum_x \sum_y xy \cdot \Pr\{X = x \text{ и } Y = y\} \\ &= \sum_x \sum_y xy \cdot \Pr\{X = x\} \Pr\{Y = y\} \\ &= \left(\sum_x x \cdot \Pr\{X = x\}\right) \left(\sum_y y \cdot \Pr\{Y = y\}\right) \\ &= \mathrm{E}[X]\,\mathrm{E}[Y] \ . \end{aligned}$$

В общем случае, если имеется n независимых в совокупности случайных величин $X_1, X_2, \ldots, X_n$, то

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\,\mathrm{E}[X_2] \cdots \mathrm{E}[X_n] \ . \tag{B.24}$$

Если случайная величина X принимает значения из множества неотрицательных целых чисел $\mathbb{N} = \{0, 1, 2, \ldots\}$, то для ее математического ожидания имеется красивая формула

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \ , \end{aligned} \tag{B.25}$$

поскольку каждый член $\Pr\{X \geq i\}$ присутствует в сумме со знаком "плюс" i раз и со знаком "минус" — $i - 1$ раз (за исключением члена $\Pr\{X \geq 0\}$, отсутствующего в сумме).

При применении выпуклой вниз функции $f(x)$ к случайной величине X ***неравенство Йенсена*** (Jensen's inequality) гласит

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X]) \tag{B.26}$$

в случае, когда математическое ожидание существует и конечно. Функция $f(x)$ называется ***выпуклой вниз*** (convex), если для всех x и y и всех $0 \leq \lambda \leq 1$ выполняется неравенство $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$.

Дисперсия и стандартное отклонение

Математическое ожидание случайной величины ничего не говорит о том, насколько сильно "разбросаны" ее значения. Например, если есть случайные величины X и Y, такие, что $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ и $\Pr\{Y = 0\} =$

$\Pr\{Y = 1\} = 1/2$, то их математические ожидания равны $1/2$, однако реальные значения Y находятся дальше от математического ожидания этой случайной величины, чем в случае X.

Понятие дисперсии случайной величины математически выражает отклонение значений случайной величины от среднего значения. ***Дисперсия*** случайной величины X с математическим ожиданием $\mathrm{E}[X]$ определяется как[5]

$$\begin{aligned} \mathrm{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\ &= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right] \\ &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}[X\mathrm{E}[X]] + \mathrm{E}^2[X] \\ &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}^2[X] + \mathrm{E}^2[X] \\ &= \mathrm{E}\left[X^2\right] - \mathrm{E}^2[X] \ . \end{aligned} \tag{B.27}$$

Чтобы пояснить равенство $\mathrm{E}[\mathrm{E}^2[X]] = \mathrm{E}^2[X]$, заметим, что, поскольку $\mathrm{E}[X]$ является действительным числом, а не случайной переменной, это значение просто равно $\mathrm{E}^2[X]$. Равенство $\mathrm{E}[X\mathrm{E}[X]] = \mathrm{E}^2[X]$ следует из уравнения (B.22) при $a = \mathrm{E}[X]$. Переписав уравнение (B.27), мы получим выражение математического ожидания квадрата случайной величины:

$$\mathrm{E}\left[X^2\right] = \mathrm{Var}[X] + \mathrm{E}^2[X] \ . \tag{B.28}$$

Дисперсия случайной величины X связана с дисперсией случайной величины aX следующим соотношением (см. упр. B.3.10):

$$\mathrm{Var}[aX] = a^2\mathrm{Var}[X] \ .$$

Если X и Y — независимые случайные величины, то

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] \ .$$

В общем случае, если n случайных величин $X_1, X_2, \ldots, X_n$ попарно независимы, то

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i] \ . \tag{B.29}$$

Стандартным отклонением (standard deviation) случайной величины X называется неотрицательный квадратный корень дисперсии X. Иногда стандартное отклонение X обозначают как σ_X или, если случайная величина очевидна из контекста, просто как σ. При использовании этого обозначения дисперсия X записывается как σ^2.

---

[5] В отечественной математической литературе для дисперсии принято обозначение $\mathrm{D}[X]$. — *Примеч. ред.*

Упражнения

В.3.1

Предположим, что бросаются две обычные шестигранные игральные кости. Чему равно математическое ожидание суммы выпавших очков? Чему равно математическое ожидание максимального из двух выпадающих чисел?

В.3.2

Массив $A[1..n]$ содержит n различных чисел в произвольном порядке; все перестановки чисел равновероятны. Чему равно математическое ожидание индекса максимального элемента массива? Чему равно математическое ожидание индекса минимального элемента массива?

В.3.3

Игрок ставит доллар на одно из чисел от 1 до 6 и выбрасывает одновременно три игральные кости. Если указанное число не выпало ни на одной из костей, игрок теряет свой доллар; если же число выпало на k костях, игрок сохраняет свой доллар и получает дополнительно k долларов. Чему равно математическое ожидание выигрыша игрока в одной партии?

В.3.4

Докажите, что если X и Y — неотрицательные случайные величины, то

$$\mathrm{E}[\max(X,Y)] \leq \mathrm{E}[X] + \mathrm{E}[Y] .$$

В.3.5 $\star$

Пусть X и Y — независимые случайные величины. Докажите, что $f(X)$ и $g(Y)$ независимы при любом выборе функций f и g.

В.3.6 $\star$

Пусть X — неотрицательная случайная величина, и математическое ожидание $\mathrm{E}[X]$ вполне определено. Докажите ***неравенство Маркова***

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t \qquad \text{(В.30)}$$

для всех $t > 0$.

В.3.7 $\star$

Пусть S — пространство выборок, а X и X' — случайные величины, такие, что $X(s) \geq X'(s)$ для всех $s \in S$. Докажите, что для произвольной действительной константы t

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\} .$$

В.3.8

Что больше — математическое ожидание квадрата случайной величины или квадрат ее математического ожидания?

В.3.9

Покажите, что для любой случайной величины X, которая принимает только значения 0 и 1, справедливо соотношение

$$\mathrm{Var}\left[X\right] = \mathrm{E}\left[X\right]\mathrm{E}\left[1 - X\right] \ .$$

В.3.10

Докажите, используя определение дисперсии (В.27), что

$$\mathrm{Var}\left[aX\right] = a^2\mathrm{Var}\left[X\right] \ .$$

В.4. Геометрическое и биномиальное распределения

Бросание симметричной монеты — пример ***испытания Бернулли*** (Bernoulli trial), которое определяется как эксперимент с двумя возможными исходами — ***успехом*** с вероятностью p и ***неудачей*** с вероятностью $q = 1 - p$. Когда речь идет об испытаниях Бернулли, то подразумевается, что испытания независимы в совокупности и (если явно не оговорено иное) что вероятность успеха в каждом испытании равна p. С испытаниями Бернулли связаны два важных распределения вероятностей: геометрическое и биномиальное.

Геометрическое распределение

Предположим, что имеется последовательность испытаний Бернулли, вероятность успеха в каждом из которых равна p, а вероятность неудачи — $q = 1 - p$. Сколько испытаний будет проведено до того, как будет достигнут успех? Пусть случайная величина X равна количеству испытаний, необходимых для достижения успеха. Тогда X принимает значения из диапазона $\{1, 2, \ldots\}$ и для $k \geq 1$

$$\Pr\left\{X = k\right\} = q^{k-1}p \ , \tag{B.31}$$

поскольку перед наступлением одного успешного испытания было выполнено $k - 1$ неудачных. Распределение вероятности, удовлетворяющее уравнению (В.31), называется ***геометрическим распределением*** (geometric distribution). На рис. В.1 показан пример такого распределения.

Полагая, что $q < 1$, можно найти математическое ожидание геометрического распределения, воспользовавшись тождеством (А.8):

$$\begin{aligned}\mathrm{E}\left[X\right] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^k\end{aligned}$$

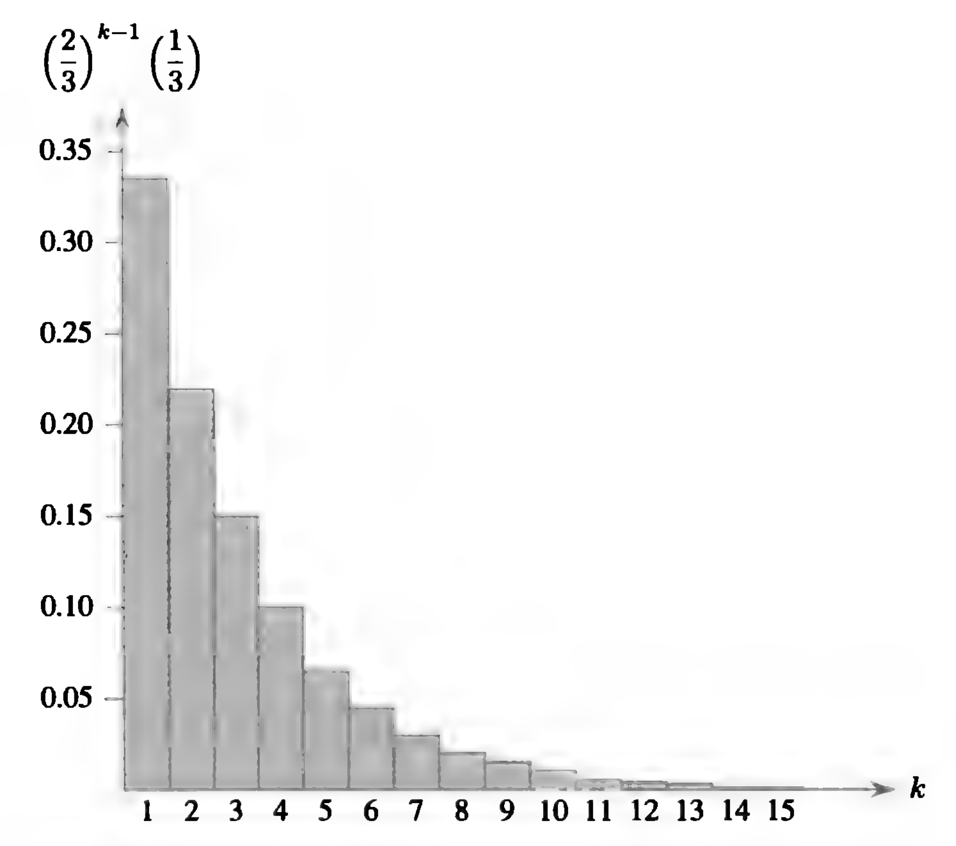


Рис. В.1. Геометрическое распределение с вероятностью успеха $p = 1/3$ и вероятностью неудачи $q = 1 - p$. Математическое ожидание распределения равно $1/p = 3$.

$$\begin{aligned} &= \frac{p}{q} \cdot \frac{q}{(1-q)^2} \\ &= \frac{p}{q} \cdot \frac{q}{p^2} \\ &= 1/p\ . \end{aligned} \tag{В.32}$$

Таким образом, в среднем нужно выполнить $1/p$ испытаний до достижения успеха (что интуитивно представляется вполне естественным). Дисперсия вычисляется аналогично, с использованием результата упр. А.1.3, и равна

$$\mathrm{Var}\,[X] = q/p^2\ . \tag{В.33}$$

В качестве примера рассмотрим бросание двух кубиков до тех пор, пока не будет получено 7 или 11 очков. Из 36 возможных исходов бросания 6 дают 7 очков, и 2 дают 11. Таким образом, вероятность успеха равна $p = 8/36 = 2/9$, так что в среднем необходимо бросить кости $1/p = 9/2 = 4.5$ раза для того, чтобы выпало 7 или 11 очков.

Биномиальное распределение

Какое количество из n испытаний Бернулли завершится успешно, если вероятность успеха равна p, а неудачи — $q = 1 - p$? Определим случайную величину X как количество успехов в n испытаниях. Тогда X принимает значения из диа-

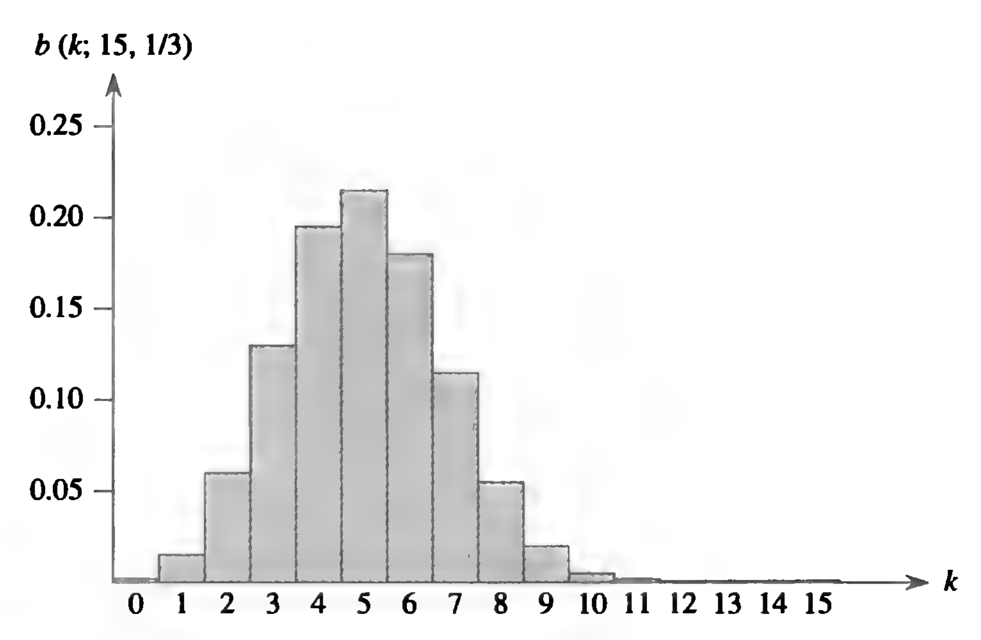


Рис. В.2. Биномиальное распределение $b(k; 15, 1/3)$, получаемое для $n = 15$ испытаний Бернулли, в каждом из которых вероятность успеха составляет $p = 1/3$. Математическое ожидание данного распределения равно $np = 5$.

пазона $\{0, 1, \ldots, n\}$ и для $k = 0, 1, \ldots, n$

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k} , \tag{В.34}$$

поскольку имеется $\binom{n}{k}$ способов выбрать k успешных испытаний из n, и вероятность каждого составляет $p^k q^{n-k}$. Распределение вероятностей (В.34) называется ***биномиальным распределением*** (binomial distribution). Для удобства для биномиального распределения используется запись

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} . \tag{В.35}$$

Пример биномиального распределения показан на рис. В.2. Название "биномиальное" связано с тем, что правая часть формулы (В.34) описывает k-й член в разложении $(p+q)^n$. Следовательно, поскольку $p + q = 1$,

$$\sum_{k=0}^{n} b(k; n, p) = 1 , \tag{В.36}$$

как того требует вторая аксиома вероятностей.

Вычислить математическое ожидание случайной величины, имеющей биномиальное распределение, можно с использованием формул (В.8) и (В.36). Пусть X — случайная величина с биномиальным распределением $b(k; n, p)$ и пусть $q = 1 - p$.

Из определения математического ожидания

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k \cdot \Pr\{X = k\} \\
&= \sum_{k=0}^{n} k \cdot b(k; n, p) \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} && \text{(согласно (В.8))} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np && \text{(согласно (В.36))} \ . \qquad \text{(В.37)}
\end{aligned}$$

Используя линейность математического ожидания, можно получить тот же результат с существенно меньшим количеством выкладок. Пусть X_i — случайная величина, описывающая количество успешных случаев в i-м испытании. Тогда $\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$ и согласно линейности математического ожидания (уравнение (В.21)) математическое ожидание числа успешных испытаний из общего количества n равно

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np \ . \qquad \text{(В.38)}
\end{aligned}$$

Тот же подход можно применить и для вычисления дисперсии распределения. Используя уравнение (В.27), получаем $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - \mathrm{E}^2[X_i]$. Поскольку X_i может принимать только значения 0 или 1, имеем $X_i^2 = X_i$, откуда $\mathrm{E}[X_i^2] = \mathrm{E}[X_i] = p$. Следовательно,

$$\mathrm{Var}[X_i] = p - p^2 = p(1-p) = pq \ . \qquad \text{(В.39)}$$

Для вычисления дисперсии X воспользуемся независимостью всех n попыток. Тогда в соответствии с (В.29)

$$\begin{aligned}\operatorname{Var}[X] &= \operatorname{Var}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \operatorname{Var}[X_i] \\ &= \sum_{i=1}^{n} pq \\ &= npq\ . \end{aligned} \tag{В.40}$$

Как видно из рис. В.2, функция биномиального распределения $b(k;n,p)$ растет с ростом k до тех пор, пока k не достигает значения np, после чего начинает уменьшаться. Мы можем доказать, что биномиальное распределение всегда ведет себя подобным образом, рассматривая отношение двух последовательных членов:

$$\begin{aligned}\frac{b(k;n,p)}{b(k-1;n,p)} &= \frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}} \\ &= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\ &= \frac{(n-k+1)p}{kq} \\ &= 1 + \frac{(n+1)p-k}{kq}\ . \end{aligned} \tag{В.41}$$

Это отношение больше единицы только тогда, когда $(n+1)p-k$ положительно. Следовательно, $b(k;n,p) > b(k-1;n,p)$ при $k < (n+1)p$ (функция распределения возрастает) и $b(k;n,p) < b(k-1;n,p)$ при $k > (n+1)p$ (функция распределения убывает). Если $k = (n+1)p$ — целое число, то $b(k;n,p) = b(k-1;n,p)$, так что функция распределения имеет два максимальных значения — при $k = (n+1)p$ и при $k-1 = (n+1)p-1 = np-q$. В противном случае она принимает максимальное значение при единственном целом значении k в диапазоне $np-q < k < (n+1)p$.

Следующая лемма дает верхнюю оценку биномиального распределения.

Лемма В.1

Пусть $n \geq 0$, $0 < p < 1$, $q = 1-p$ и $0 \leq k \leq n$. Тогда

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\ .$$

Доказательство. Используя уравнение (В.6), получаем

$$\begin{aligned} b(k;n,p) &= \binom{n}{k} p^k q^{n-k} \\ &\le \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} . \end{aligned}$$

■

Упражнения

В.4.1

Убедитесь в выполнении аксиомы 2 для геометрического распределения.

В.4.2

Сколько раз в среднем нужно бросить шесть симметричных монет для получения трех орлов и трех решек?

В.4.3

Покажите, что $b(k;n,p) = b(n-k;n,q)$, где $q = 1 - p$.

В.4.4

Покажите, что значение максимума функции биномиального распределения $b(k;n,p)$ приближенно равно $1/\sqrt{2\pi npq}$, где $q = 1 - p$.

В.4.5 ★

Покажите, что вероятность не получить ни одного успешного исхода в серии из n испытаний Бернулли с вероятностью успеха $p = 1/n$ составляет приблизительно $1/e$. Покажите также, что вероятность получения ровно одного успешного исхода также составляет примерно $1/e$.

В.4.6 ★

Профессора Однокаменяк и Айнштайн бросают симметричную монету n раз. Покажите, что вероятность того, что они получат одинаковое количество орлов, равна $\binom{2n}{n}/4^n$. (*Указание:* считайте успехом выпадение орла у профессора Однокаменяка, и выпадение решки у профессора Айнштайна.) Воспользуйтесь своей аргументацией для проверки тождества

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} .$$

В.4.7 ★

Покажите, что для $0 \le k \le n$,

$$b(k;n,1/2) \le 2^{n\,H(k/n)-n} ,$$

где $H(x)$ — энтропийная функция (В.7).

B.4.8 ★

Рассмотрим n испытаний Бернулли, где p_i — вероятность успеха в i-м испытании для $i = 1, 2, \ldots, n$, а X — случайная величина, равная общему количеству успехов. Пусть $p \geq p_i$ для всех $i = 1, 2, \ldots, n$. Докажите, что для $1 \leq k \leq n$

$$\Pr\{X < k\} \geq \sum_{i=0}^{k-1} b(i; n, p) \ .$$

B.4.9 ★

Пусть X — случайная величина, равная общему количеству успехов в множестве A из n испытаний Бернулли, а p_i — вероятность успеха в i-м испытании. Пусть X' — аналогичная случайная величина, равная общему количеству успехов в множестве A' из n испытаний Бернулли, где $p'_i \geq p_i$ — вероятность успеха в i-м испытании. Докажите, что для $0 \leq k \leq n$

$$\Pr\{X' \geq k\} \geq \Pr\{X \geq k\} \ .$$

(*Указание:* покажите, как получить результаты испытаний Бернулли A' из эксперимента, включающего испытания A, а затем воспользуйтесь результатом упр. B.3.7.)

★ B.5. Хвосты биномнального распределения

Зачастую во многих задачах требуется определить не вероятность того, что в n испытаниях Бернулли будет получено ровно k успешных исходов, а вероятность того, что будет получено не более (или не менее) k успешных исходов. В этом разделе мы рассмотрим ***хвосты*** (tails) биномиального распределения, т.е. области распределения $b(k; n, p)$, далекие от среднего значения np, и найдем некоторые важные оценки для них.

Мы рассмотрим правый хвост распределения $b(k; n, p)$; левый хвост получается при простой взаимной замене успешных исходов неудачами.

Теорема B.2

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 \leq k \leq n$ вероятность того, что будет получено как минимум k успешных исходов, равна

$$\begin{aligned}\Pr\{X \geq k\} &= \sum_{i=k}^{n} b(i; n, p) \\ &\leq \binom{n}{k} p^k \ .\end{aligned}$$

Доказательство. Для множества $S \subseteq \{1, 2, \ldots, n\}$ обозначим через A_S событие, которое заключается в том, что i-я попытка успешна для всех $i \in S$. Понятно, что если $|S| = k$, то $\Pr\{A_S\} = p^k$. Имеем

$$\begin{aligned}
\Pr\{X \geq k\} &= \Pr\{\text{существует } S \subseteq \{1, 2, \ldots, n\} : |S| = k \text{ и } A_S\} \\
&= \Pr\left\{\bigcup_{S \subseteq \{1,2,\ldots,n\}:|S|=k} A_S\right\} \\
&\leq \sum_{S \subseteq \{1,2,\ldots,n\}:|S|=k} \Pr\{A_S\} \qquad \text{(согласно (В.19))} \\
&= \binom{n}{k} p^k .
\end{aligned}$$

■

Приведенное далее следствие просто переформулирует эту теорему для левого хвоста биномиального распределения. Все доказательства переформулированных таким образом (для противоположного хвоста) теорем далее в этом приложении предлагаются читателю в качестве самостоятельного упражнения.

Следствие В.3

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 \leq k \leq n$ вероятность того, что будет получено не более k успешных исходов, равна

$$\begin{aligned}
\Pr\{X \leq k\} &= \sum_{i=0}^{k} b(i; n, p) \\
&\leq \binom{n}{n-k} (1-p)^{n-k} \\
&= \binom{n}{k} (1-p)^{n-k} .
\end{aligned}$$

■

Следующая рассматриваемая оценка относится к левому хвосту биномиального распределения. Следствие из нее демонстрирует, что вдалеке от среднего значения левый хвост уменьшается экспоненциально.

Теорема В.4

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1 - p$. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 < k < np$ вероятность того, что будет получено менее k

успешных исходов, равна

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i; n, p) \\ &< \frac{kq}{np - k}\, b(k; n, p) \ .\end{aligned}$$

Доказательство. Ограничим ряд $\sum_{i=0}^{k-1} b(i; n, p)$ геометрическим рядом с использованием методики из раздела А.2, с. 1204. Для $i = 1, 2, \ldots, k$ из уравнения (B.41) имеем

$$\begin{aligned}\frac{b(i-1; n, p)}{b(i; n, p)} &= \frac{iq}{(n-i+1)p} \\ &< \frac{iq}{(n-i)p} \\ &\le \frac{kq}{(n-k)p} \ .\end{aligned}$$

Если положим

$$\begin{aligned}x &= \frac{kq}{(n-k)p} \\ &< \frac{kq}{(n-np)p} \\ &= \frac{kq}{nqp} \\ &= \frac{k}{np} \\ &< 1 \ ,\end{aligned}$$

то получим

$$b(i-1; n, p) < x\, b(i; n, p)$$

для $0 < i \le k$. Итеративно применив это неравенство $k - i$ раз, получим

$$b(i; n, p) < x^{k-i}\, b(k; n, p)$$

для $0 \le i < k$, а следовательно,

$$\begin{aligned}\sum_{i=0}^{k-1} b(i; n, p) &< \sum_{i=0}^{k-1} x^{k-i} b(k; n, p) \\ &< b(k; n, p) \sum_{i=1}^{\infty} x^i\end{aligned}$$

$$= \frac{x}{1-x}\, b(k;n,p)$$
$$= \frac{kq}{np-k}\, b(k;n,p)\ . \qquad \blacksquare$$

Следствие В.5

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1 - p$. Тогда для $0 < k \leq np/2$ вероятность того, что будет получено менее k успешных исходов, составляет менее половины от вероятности получения менее $k+1$ успешного исхода.

Доказательство. Поскольку $k \leq np/2$, имеем

$$\begin{aligned} \frac{kq}{np-k} &\leq \frac{(np/2)q}{np-(np/2)} \\ &= \frac{(np/2)q}{np/2} \\ &\leq 1\ , \end{aligned} \tag{B.42}$$

так как $q \leq 1$. Пусть X — случайная величина, равная общему количеству успешных исходов. Из теоремы В.4 и неравенства (В.42) следует, что вероятность получить менее k успешных исходов равна

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i;n,p) < b(k;n,p)\ .$$

Таким образом,

$$\begin{aligned} \frac{\Pr\{X<k\}}{\Pr\{X<k+1\}} &= \frac{\sum_{i=0}^{k-1} b(i;n,p)}{\sum_{i=0}^{k} b(i;n,p)} \\ &= \frac{\sum_{i=0}^{k-1} b(i;n,p)}{\sum_{i=0}^{k-1} b(i;n,p) + b(k;n,p)} \\ &< 1/2\ , \end{aligned}$$

поскольку $\sum_{i=0}^{k-1} b(i;n,p) < b(k;n,p)$. ■

Оценка для правого хвоста выполняется аналогично. Ее доказательство оставлено читателю в качестве упр. В.5.2.

Следствие В.6

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $np < k < n$ вероятность

более чем k успешных исходов равна

$$\Pr\{X > k\} = \sum_{i=k+1}^{n} b(i; n, p) < \frac{(n-k)p}{k-np} b(k; n, p) .$$

■

Следствие В.7

Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1 - p$. Тогда для $(np + n)/2 < k < n$ вероятность более чем k успешных исходов не превышает половины вероятности более чем $k - 1$ успешных исходов.

В следующей теореме рассматриваются n испытаний Бернулли; вероятность успеха в i-м испытании составляет p_i. Как видно из следствия из данной теоремы, ее можно использовать для оценки правого хвоста биномиального распределения, положив $p_i = p$ для всех испытаний.

Теорема В.8

Рассмотрим последовательность из n испытаний Бернулли, в которой в i-м испытании ($i = 1, 2, \ldots, n$) вероятность успеха равна p_i, а неудачи — $q_i = 1 - p_i$. Пусть X — случайная величина, равная общему количеству успешных исходов, а $\mu = \mathrm{E}[X]$. Тогда для $r > \mu$

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r .$$

Доказательство. Поскольку при любом $\alpha > 0$ функция $e^{\alpha x}$ строго возрастающая по x,

$$\Pr\{X - \mu \geq r\} = \Pr\left\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\right\} , \tag{B.43}$$

где значение α будет определено позже. Использовав неравенство Маркова (B.30), получим

$$\Pr\left\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\right\} \leq \mathrm{E}\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r} . \tag{B.44}$$

Теперь нужно оценить величину $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ и подставить подходящее значение α в неравенство (B.44). Начнем с вычисления $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$. Используя индикаторные случайные величины (см. раздел 5.2), положим $X_i = \mathrm{I}\{i\text{-е испытание Бернулли успешно}\}$ при $i = 1, 2, \ldots, n$; т.е. X_i представляет собой случайную величину, которая равна 1, если i-е испытание Бернулли успешно, и 0, если оно неудачно. Таким образом,

$$X = \sum_{i=1}^{n} X_i ,$$

и согласно линейности математического ожидания

$$\mu = \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p_i ,$$

откуда вытекает

$$X - \mu = \sum_{i=1}^{n}(X_i - p_i) .$$

Чтобы вычислить $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$, подставим в это выражение найденное значение $X - \mu$ и получим

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \mathrm{E}\left[e^{\alpha\sum_{i=1}^{n}(X_i-p_i)}\right] \\
&= \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha(X_i-p_i)}\right] \\
&= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i-p_i)}\right] ,
\end{aligned}$$

что следует из (В.24), поскольку случайные величины X_i являются независимыми в совокупности, что влечет независимость в совокупности случайных величин $e^{\alpha(X_i-p_i)}$ (см. упр. В.3.5). Из определения математического ожидания

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X_i-p_i)}\right] &= e^{\alpha(1-p_i)}p_i + e^{\alpha(0-p_i)}q_i \\
&= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\
&\le p_i e^{\alpha} + 1 \qquad \text{(В.45)} \\
&\le \exp(p_i e^{\alpha}) ,
\end{aligned}$$

где $\exp(x)$ обозначает экспоненциальную функцию: $\exp(x) = e^x$. (Неравенство (В.45) следует из неравенств $\alpha > 0$, $q_i \le 1$, $e^{\alpha q_i} \le e^{\alpha}$ и $e^{-\alpha p_i} \le 1$, а последняя строка — из неравенства (3.12).) Следовательно,

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i-p_i)}\right] \\
&\le \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\
&= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) \\
&= \exp(\mu e^{\alpha}) , \qquad \text{(В.46)}
\end{aligned}$$

поскольку $\mu = \sum_{i=1}^{n} p_i$. Следовательно, из уравнения (В.43) и неравенств (В.44) и (В.46) вытекает, что

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^{\alpha} - \alpha r) \ . \qquad \text{(В.47)}$$

Выбрав $\alpha = \ln(r/\mu)$ (см. упр. В.5.7), получим

$$\begin{aligned}\Pr\{X - \mu \geq r\} &\leq \exp(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)) \\ &= \exp(r - r\ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r\end{aligned}$$

■

Применив теорему В.8 к последовательности n испытаний Бернулли с равной вероятностью успеха, получаем оценку для правого хвоста биномиального распределения.

Следствие В.9

Рассмотрим последовательность n испытаний Бернулли с равной вероятностью успеха p и неудачи $q = 1 - p$ в каждом испытании. Тогда для $r > np$

$$\begin{aligned}\Pr\{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \\ &\leq \left(\frac{npe}{r}\right)^r\end{aligned}$$

Доказательство. Согласно (В.37) получаем $\mu = \mathrm{E}[X] = np$. ■

Упражнения

В.5.1 ★

Что менее вероятно: при бросании симметричной монеты n раз не получить ни одного орла или получить менее n орлов при бросании монеты $4n$ раз?

В.5.2 ★

Докажите следствия В.6 и В.7.

В.5.3 ★

Покажите, что

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

для всех $a > 0$ и всех k, таких, что $0 < k < na/(a+1)$.

В.5.4 ★

Докажите, что если $0 < k < np$, где $0 < p < 1$ и $q = 1 - p$, то

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

В.5.5 ★

Воспользуйтесь теоремой В.8, чтобы показать, что

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r$$

для $r > n - \mu$. Аналогично воспользуйтесь следствием В.9, чтобы показать, что

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r$$

для $r > n - np$.

В.5.6 ★

Рассмотрим последовательность n испытаний Бернулли, в которой в i-м испытании ($i = 1, 2, \ldots, n$) вероятность успеха равна p_i, а неудачи — $q_i = 1 - p_i$. Пусть X — случайная величина, равная общему количеству успешных исходов, а $\mu = \mathrm{E}[X]$. Покажите, что для $r \geq 0$

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n} .$$

(*Указание:* Докажите, что $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}$. Затем следуйте схеме доказательства теоремы В.8, используя это неравенство вместо неравенства (В.45).)

В.5.7 ★

Покажите, что выбор $\alpha = \ln(r/\mu)$ минимизирует величину правой стороны неравенства (В.47).

Задачи

В.1. Шары и корзины

В этой задаче рассмотрим размещение n шаров по b различным корзинам.

а. Предположим, что все n шаров различны, а их порядок в корзине не имеет значения. Докажите, что имеется b^n способов размещения шаров в корзинах.

б. Предположим, что все n шаров различны, а их порядок в корзине существенен. Докажите, что имеется ровно $(b + n - 1)!/(b - 1)!$ способов размещения шаров в корзинах. (*Указание:* подсчитайте количество способов разместить в ряд n различных шаров и $b - 1$ неразличимых разделителей.)

в. Предположим, что все шары идентичны, а следовательно, их порядок в корзине не имеет значения. Покажите, что количество способов, которыми можно разместить шары в корзинах, равно $\binom{b+n-1}{n}$. (*Указание:* воспользуйтесь тем же способом, что и в части (б), только теперь шары также неразличимы.)

г. Покажите, что если шары идентичны и ни в одной корзине не может находиться больше одного шара, то разместить шары в корзинах можно $\binom{b}{n}$ способами.

д. Покажите, что если шары идентичны и ни одна корзина не должна остаться пустой, то разместить шары в корзинах можно $\binom{n-1}{b-1}$ способами в предположении, что $n \geq b$.

Заключительные замечания

Общие методы решения вероятностных задач впервые обсуждались в знаменитой переписке Б. Паскаля (B. Pascal) и П. Ферма (P. de Fermat), начавшейся в 1654 году, и в книге Х. Гюйгенса (C. Huygens, 1657). Строгая теория вероятности началась с работ Я. Бернулли (J. Bernoulli, 1713) и А. де Муавра (A. de Moivre, 1730). Дальнейшее развитие теории вероятности связано с именами П. Лапласа (P.S. de Laplace), С.-Д. Пуассона (S.-D. Poisson) и К.Ф. Гаусса (C.F. Gauss).

Суммы случайных величин исследовались П.Л. Чебышевым и А.А. Марковым. Аксиоматизация теории вероятности была выполнена А.Н. Колмогоровым в 1933 году. Оценки хвостов распределений приведены в работах Чернова (Chernoff) [65] и Хеффдинга (Hoeffding) [172]. Важные результаты о случайных комбинаторных структурах принадлежат П. Эрдешу (P. Erdős).

Материалы по элементарной комбинаторике можно найти в книгах Кнута (Knuth) [208][6] и Лю (Liu) [236]; по теории вероятности — в книгах Биллингсли (Billingsley) [45], Чанга (Chung) [66], Дрейка (Drake) [94], Феллера (Feller) [103] и Розанова (Rozanov) [298].

[6] Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: И.Д. "Вильямс", 2000. Кроме того, уже после написания данной книги вышел очередной том "*Искусства программирования*", полностью посвященный вопросам комбинаторики: Д. Кнут. *Искусство программирования, т. 4, А. Комбинаторные алгоритмы, часть 1.* — М.: И.Д. "Вильямс", 2013.

Приложение Г. Матрицы

С матрицами приходится сталкиваться в множестве различных приложений, включая (но не ограничиваясь) научные вычисления. Если вы встречались с матрицами ранее, значит, бо́льшая часть приведенного здесь материала будет вам знакома, но кое-что вы можете увидеть впервые. В разделе Г.1 рассматриваются основные определения и операции, а в разделе Г.2 — некоторые свойства матриц.

Г.1. Матрицы и матричные операции

В этом разделе мы рассмотрим основные концепции теории матриц и некоторые их фундаментальные свойства.

Матрицы и векторы

Матрица (matrix) представляет собой прямоугольный массив чисел. Например,

$$
\begin{aligned}
A &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad (\text{Г.1})
\end{aligned}
$$

представляет собой матрицу $A = (a_{ij})$ размером 2×3, где для $i = 1, 2$ и $j = 1, 2, 3$ элемент матрицы, находящийся в строке i и столбце j, обозначается как a_{ij}. Прописные буквы используются для обозначения матриц, а соответствующие строчные — для их элементов. Множество всех $m \times n$-матриц действительных чисел обозначается как $\mathbb{R}^{m \times n}$ и в общем случае множество $m \times n$ матриц с элементами, выбранными из множества S, — как $S^{m \times n}$.

Транспонированная (transpose) матрица A^{T} получается из матрицы A путем обмена местами ее строк и столбцов. Так, для матрицы A из (Г.1)

$$
A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} .
$$

Вектор (vector) представляет собой одномерный массив чисел. Например,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$$

является вектором размером 3. Для обозначения векторов мы используем строчные буквы и обозначаем i-й элемент вектора x как x_i. Стандартной формой вектора будем считать ***вектор-столбец*** (column vector), эквивалентный матрице $n \times 1$. Соответствующий ***вектор-строка*** (row vector) получается путем транспонирования вектора-столбца:

$$x^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & 5 \end{pmatrix}.$$

Единичным вектором (unit vector) e_i называется вектор, i-й элемент которого равен единице, а все остальные элементы равны нулю. Обычно размер единичного вектора ясен из контекста.

Нулевая матрица (zero matrix) — это матрица, все элементы которой равны нулю. Такая матрица часто записывается просто как 0, поскольку неоднозначность между числом 0 и нулевой матрицей легко разрешается с помощью контекста. Если размер нулевой матрицы не указан, то он также выводится из контекста.

Квадратные матрицы

Часто приходится иметь дело с ***квадратными*** (square) матрицами размером $n \times n$. Некоторые из квадратных матриц представляют особый интерес.

1. ***Диагональная матрица*** (diagonal matrix) обладает тем свойством, что $a_{ij} = 0$ при $i \neq j$. Поскольку все недиагональные элементы такой матрицы равны нулю, диагональную матрицу можно определить путем перечисления ее элементов вдоль диагонали:

$$\operatorname{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix}.$$

2. ***Единичная матрица*** (identity matrix) I_n размером $n \times n$ представляет собой диагональную матрицу, все диагональные элементы которой равны единице:

$$\begin{aligned} I_n &= \operatorname{diag}(1, 1, \ldots, 1) \\ &= \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}. \end{aligned}$$

Если используется обозначение I без индекса, размер единичной матрицы определяется из контекста. Заметим, что i-м столбцом единичной матрицы является единичный вектор e_i.

3. Элементы ***трехдиагональной матрицы*** (tridiagonal matrix) T обладают тем свойством, что если $|i-j|>1$, то $t_{ij}=0$. Ненулевые элементы такой матрицы располагаются на главной диагонали, непосредственно над ней ($t_{i,i+1}$ для $i = 1,2,\ldots,n-1$) и непосредственно под ней ($t_{i+1,i}$ для $i=1,2,\ldots,n-1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix} .$$

4. ***Верхнетреугольной матрицей*** (upper-triangular matrix) U называется матрица, все элементы которой ниже диагонали равны нулю ($u_{ij}=0$ при $i>j$):

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix} .$$

   Верхнетреугольная матрица является ***единичной верхнетреугольной матрицей*** (unit upper-triangular), если все ее диагональные элементы равны единице.

5. ***Нижнетреугольной матрицей*** (lower-triangular matrix) L называется матрица, все элементы которой выше диагонали равны нулю ($l_{ij}=0$ при $i<j$):

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix} .$$

   Нижнетреугольная матрица является ***единичной нижнетреугольной матрицей*** (unit lower-triangular), если все ее диагональные элементы равны единице.

6. ***Матрица перестановки*** (permutation matrix) P имеет в каждой строке и столбце ровно по одной единице, а на всех прочих местах располагаются

нули. Примером матрицы перестановки может служить матрица

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} .$$

Такая матрица называется матрицей перестановки, потому что умножение вектора x на матрицу перестановки приводит к перестановке элементов вектора. В упр. Г.1.4 рассматриваются дополнительные свойства матриц перестановок.

7. ***Симметричная матрица*** (symmetric matrix) A удовлетворяет условию $A = A^{\mathrm{T}}$. Например, матрица

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

является симметричной.

Основные матричные операции

Элементами матриц и векторов служат элементы некоторой числовой системы, такие как действительные числа, комплексные числа или, например, целые числа по модулю простого числа. Числовая система определяет, каким образом должны складываться и перемножаться числа. Эти определения можно распространить и на матрицы.

Мы определяем ***сложение матриц*** (matrix addition) следующим образом. Если $A = (a_{ij})$ и $B = (b_{ij})$ – матрицы размером $m \times n$, то их суммой является матрица $C = (c_{ij}) = A + B$ размером $m \times n$, определяемая соотношениями

$$c_{ij} = a_{ij} + b_{ij}$$

$i = 1, 2, \ldots, m$ и $j = 1, 2, \ldots, n$. Другими словами, сложение матриц выполняется поэлементно. Нулевая матрица является единичным элементом по отношению к сложению матриц:

$$A + 0 = A = 0 + A .$$

Если λ — число, а $A = (a_{ij})$ — матрица, то соотношение $\lambda A = (\lambda a_{ij})$ определяет ***скалярное произведение*** (scalar multiple) матрицы на число, которое также выполняется поэлементно. Частным случаем скалярного произведения является умножение на -1, которое дает ***противоположную*** (negative) матрицу $-1 \cdot A = -A$, обладающую тем свойством, что ij-й элемент $-A$ равен $-a_{ij}$. Таким образом,

$$A + (-A) = 0 = (-A) + A .$$

Понятие противоположной матрицы используется при определении ***вычитания матриц*** (matrix subtraction): $A - B = A + (-B)$.

Матричное умножение (matrix multiplication) определяется следующим образом. Матрицы A и B могут быть перемножены, если они ***совместимы*** (compatible) в том смысле, что число столбцов A равно числу строк B (в общем случае выражение, содержащее матричное произведение AB, всегда подразумевает совместимость матриц A и B). Если $A = (a_{ij})$ — матрица размером $m \times n$, а $B = (b_{ij})$ — матрица размером $n \times p$, то их произведение $C = AB$ представляет собой матрицу $C = (c_{ij})$ размером $m \times p$, элементы которой определяются уравнением

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{Г.2}$$

для $i = 1, 2, \ldots, m$ и $j = 1, 2, \ldots, p$. Процедура SQUARE-MATRIX-MULTIPLY из раздела 4.2 реализует матричное умножение квадратных матриц непосредственно по формуле (Г.2). Для умножения матриц размером $n \times n$ процедура SQUARE-MATRIX-MULTIPLY выполняет n^3 умножений и $n^2(n-1)$ сложений, так что время ее работы равно $\Theta(n^3)$.

Матрицы обладают многими (но не всеми) алгебраическими свойствами, присущими обычным числам. Единичная матрица является единичным элементом по отношению к умножению:

$$I_m A = A I_n = A$$

для любой матрицы A размером $m \times n$. Умножение на нулевую матрицу дает нулевую матрицу:

$$A\,0 = 0\ .$$

Умножение матриц ассоциативно:

$$A(BC) = (AB)C$$

для любых совместимых матриц A, B и C. Умножение матриц дистрибутивно относительно сложения:

$$A(B + C) = AB + AC\ ,$$
$$(B + C)D = BD + CD\ .$$

Для $n > 1$ умножение матриц размером $n \times n$ не коммутативно. Например, если $A = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ и $B = \left(\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right)$, то

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

и

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\ .$$

Произведения матрицы и вектора или двух векторов вычисляются путем представления вектора как матрицы размером $n \times 1$ (или $1 \times n$ в случае вектора-строки).

Таким образом, если A — матрица размером $m \times n$, а x — вектор размером n, то Ax является вектором размером m. Если x и y — векторы размером n, то произведение

$$x^{\mathrm{T}}y = \sum_{i=1}^{n} x_i y_i$$

представляет собой число (в действительности — матрицу размером 1×1), называемое ***скалярным произведением*** (inner product) векторов x и y. Матрица $Z = xy^{\mathrm{T}}$ размером $n \times n$ с элементами $z_{ij} = x_i y_j$ называется ***тензорным произведением*** (outer product) этих же векторов. (***Евклидова***) ***норма*** ((euclidean) norm) $\|x\|$ вектора x размером n определяется соотношением

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \\ &= (x^{\mathrm{T}}x)^{1/2} . \end{aligned}$$

Таким образом, норма x — это длина вектора в n-мерном евклидовом пространстве.

Упражнения

Г.1.1

Покажите, что если A и B являются симметричными матрицами размером $n \times n$, то то же справедливо и в отношении матриц $A + B$ и $A - B$.

Г.1.2

Докажите, что $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ и что $A^{\mathrm{T}}A$ всегда является симметричной матрицей.

Г.1.3

Докажите, что произведение двух нижнетреугольных матриц является нижнетреугольной матрицей.

Г.1.4

Докажите, что если P представляет собой матрицу перестановки размером $n \times n$, а A является матрицей размером $n \times n$, то матричное произведение PA представляет собой матрицу A с переставленными строками, а матричное произведение AP представляет собой матрицу A с переставленными строками. Докажите, что произведение двух матриц перестановки является матрицей перестановки.

Г.2. Основные свойства матриц

В этом разделе будут определены некоторые основные свойства матриц: обратимость, линейная зависимость и независимость, ранги и детерминанты. Мы также изучим класс положительно определенных матриц.

Обратные матрицы, ранги и детерминанты

Матрицей, ***обратной*** (inverse) к данной матрице A размером $n \times n$, является матрица размером $n \times n$, обозначаемая как A^{-1} (если таковая существует), такая, что $AA^{-1} = I_n = A^{-1}A$, например

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} .$$

Многие ненулевые квадратные матрицы размером $n \times n$ не имеют обратных матриц. Матрица, для которой не существует обратная матрица, называется ***необращаемой*** (noninvertible) или ***вырожденной*** (singular). Вот пример ненулевой вырожденной матрицы:

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} .$$

Если матрица имеет обратную матрицу, она называется ***обращаемой*** (invertible) или ***невырожденной*** (nonsingular). Если обратная матрица существует, то она единственная (см. упр. Г.2.1). Если A и B — невырожденные матрицы размером $n \times n$, то

$$(BA)^{-1} = A^{-1}B^{-1} .$$

Операция обращения коммутативна с операцией транспонирования:

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1} .$$

Векторы $x_1, x_2, \ldots, x_n$ ***линейно зависимы*** (linearly dependent), если существуют коэффициенты $c_1, c_2, \ldots, c_n$, среди которых не все нулевые, такие, что $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. Например, векторы-строки $x_1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$, $x_2 = \begin{pmatrix} 2 & 6 & 4 \end{pmatrix}$ и $x_3 = \begin{pmatrix} 4 & 11 & 9 \end{pmatrix}$ линейно зависимы, поскольку $2x_1 + 3x_2 - 2x_3 = 0$. Если векторы не являются линейно зависимыми, они называются ***линейно независимыми*** (linearly independent). Например, столбцы единичной матрицы линейно независимы.

Столбцовым рангом (column rank) ненулевой матрицы A размером $m \times n$ называется размер наибольшего множества линейно независимых столбцов A. Аналогично ***строчным рангом*** (row rank) ненулевой матрицы A размером $m \times n$ называется размер наибольшего множества линейно независимых строк A. Фундаментальным свойством любой матрицы A является равенство ее строчного и столбцового рангов, так что мы можем говорить просто о ***ранге*** (rank) матрицы. Ранг матрицы размером $m \times n$ представляет собой целое число от нуля до $\min(m, n)$ включительно (ранг нулевой матрицы равен нулю, ранг единичной матрицы размером $n \times n$ равен n). Другое эквивалентное (и зачастую более полезное) определение ранга ненулевой матрицы A размером $m \times n$ — это наименьшее число r, такое, что существуют матрицы B и C размером соответственно $m \times r$ и $r \times n$, такие, что

$$A = BC .$$

Квадратная матрица размером $n \times n$ имеет ***полный ранг*** (full rank), если ее ранг равен n. Матрица размером $m \times n$ имеет ***полный столбцовый ранг*** (full column rank), если ее ранг равен n. Фундаментальное свойство рангов приведено в следующей теореме.

Теорема Г.1

Квадратная матрица имеет полный ранг тогда и только тогда, когда она является невырожденной. ■

Ненулевой вектор x, такой, что $Ax = 0$, называется ***аннулирующим вектором*** (null vector) матрицы A. Приведенная далее теорема (доказательство которой оставлено в качестве упр. Г.2.7) и ее следствие связывают столбцовый ранг и вырожденность с аннулирующим вектором.

Теорема Г.2

Матрица A имеет полный столбцовый ранг тогда и только тогда, когда для нее не существует аннулирующего вектора. ■

Следствие Г.3

Квадратная матрица A является вырожденной тогда и только тогда, когда она имеет аннулирующий вектор. ■

ij-минором матрицы A размером $n \times n$ $(n > 1)$ называется матрица $A_{[ij]}$ размером $(n-1) \times (n-1)$, получаемая из A удалением i-й строки и j-го столбца. ***Определитель***, или ***детерминант*** (determinant), матрицы A размером $n \times n$ можно определить рекурсивно с помощью миноров следующим образом:

$$\det(A) = \begin{cases} a_{11}\ , & \text{если } n = 1\ , \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{[1j]})\ , & \text{если } n > 1\ . \end{cases}$$

Множитель $(-1)^{i+j} \det(A_{[ij]})$ называется ***алгебраическим дополнением*** (cofactor) элемента a_{ij}.

В приведенных ниже теоремах, доказательство которых здесь опущено, описываются фундаментальные свойства определителей.

Теорема Г.4 (Свойства определителя)

Определитель квадратной матрицы A обладает следующими свойствами.

- Если любая строка или любой столбец A нулевой, то $\det(A) = 0$.
- Если все элементы одного произвольного столбца (или строки) матрицы умножаются на λ, то ее определитель также умножается на λ.
- Определитель матрицы A остается неизменным, если все элементы одной строки (или столбца) прибавить к элементам другой строки (столбца).

- Определитель матрицы A равен определителю транспонированной матрицы A^{T}.
- Определитель матрицы A умножается на -1, если обменять местами любые два ее столбца (или строки).

Кроме того, для любых квадратных матриц A и B справедливо соотношение $\det(AB) = \det(A)\det(B)$. ■

Теорема Г.5

Матрица A размером $n \times n$ вырождена тогда и только тогда, когда $\det(A) = 0$. ■

Положительно определенные матрицы

Во многих приложениях важную роль играют положительно определенные матрицы. Матрица A размером $n \times n$ является ***положительно определенной*** (positive-definite), если $x^{\mathrm{T}}Ax > 0$ для любого ненулевого вектора x размером n. Например, единичная матрица положительно определенная, поскольку для произвольного ненулевого вектора $x = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}^{\mathrm{T}}$

$$\begin{aligned} x^{\mathrm{T}} I_n x &= x^{\mathrm{T}} x \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0 \ . \end{aligned}$$

Матрицы, возникающие в различных приложениях, зачастую оказываются положительно определенными в силу следующей теоремы.

Теорема Г.6

Для произвольной матрицы A с полным столбцовым рангом матрица $A^{\mathrm{T}}A$ положительно определена.

Доказательство. Необходимо показать, что $x^{\mathrm{T}}(A^{\mathrm{T}}A)x > 0$ для любого ненулевого вектора x. Для произвольного вектора x

$$\begin{aligned} x^{\mathrm{T}}(A^{\mathrm{T}}A)x &= (Ax)^{\mathrm{T}}(Ax) && \text{(согласно упр. Г.1.2)} \\ &= \|Ax\|^2 \ . \end{aligned}$$

Заметим, что $\|Ax\|^2$ представляет собой просто сумму квадратов элементов вектора Ax. Таким образом, $\|Ax\|^2 \geq 0$. Если $\|Ax\|^2 = 0$, все элементы вектора Ax равны нулю, т.е. $Ax = 0$. Поскольку матрица A имеет полный столбцовый ранг, из $Ax = 0$ согласно теореме Г.2 следует, что $x = 0$. Следовательно, матрица $A^{\mathrm{T}}A$ положительно определенная. ■

Другие свойства положительно определенных матриц рассматриваются в разделе 28.3.

Упражнения

Г.2.1

Докажите единственность обратной матрицы, т.е. что если B и C обратны к A, то $B = C$.

Г.2.2

Докажите, что определители нижне- и верхнетреугольной матриц равны произведению их диагональных элементов. Докажите, что матрица, обратная к нижнетреугольной матрице (если таковая существует), также является нижнетреугольной.

Г.2.3

Докажите, что если P является матрицей перестановок, то матрица P обратима, обратной к ней является матрица P^{T} и P^{T} представляет собой матрицу перестановок.

Г.2.4

Пусть A и B — матрицы размером $n \times n$, такие, что $AB = I$. Докажите, что если A' получена из A путем прибавления к строке i строки j, то обратную к A' матрицу B' можно получить, вычтя в матрице B столбец i из столбца j.

Г.2.5

Пусть A — невырожденная матрица размером $n \times n$ с комплексными элементами. Покажите, что все элементы матрицы A^{-1} вещественны тогда и только тогда, когда вещественны все элементы матрицы A.

Г.2.6

Покажите, что если A — невырожденная симметричная матрица размером $n \times n$, то матрица A^{-1} симметрична. Покажите, что если B — произвольная матрица размером $m \times n$, то матрица размером $m \times m$, полученная в результате умножения BAB^{T}, симметрична.

Г.2.7

Докажите теорему Г.2, т.е. покажите, что матрица A имеет полный столбцовый ранг тогда и только тогда, когда из $Ax = 0$ следует $x = 0$. (*Указание:* запишите условие линейной зависимости одного столбца от остальных в виде матрично-векторного уравнения.)

Г.2.8

Докажите, что для любых совместимых матриц A и B

$$\mathrm{rank}(AB) \leq \min(\mathrm{rank}(A), \mathrm{rank}(B)) \ ,$$

причем равенство достигается, если либо A, либо B — невырожденная квадратная матрица. (*Указание:* воспользуйтесь альтернативным определением ранга матрицы.)

Задачи

Г.1. Матрица Вандермонда

Докажите, что для заданных чисел $x_0, x_1, \ldots, x_{n-1}$ определитель ***матрицы Вандермонда*** (Vandermonde matrix)

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

равен

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j < k \le n-1} (x_k - x_j) \; .$$

(*Указание:* умножьте столбец i на $-x_0$ и прибавьте к столбцу $i + 1$ для $i = n-1, n-2, \ldots, 1$, а затем примените индукцию.)

Г.2. Перестановки, определенные матрично-векторным умножением над полем $GF(2)$

Один из классов перестановок целых чисел из множества $S_n = \{0, 1, 2, \ldots, 2^n - 1\}$ определяется матричным умножением над полем $GF(2)$. Для каждого целого числа x из S_n мы рассматриваем его двоичное представление как n-битовый вектор

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} ,$$

где $x = \sum_{i=0}^{n-1} x_i 2^i$. Если A представляет собой матрицу размером $n \times n$, в которой каждый элемент равен либо нулю, либо единице, то можно определить отображение-перестановку каждого значения x из S_n на число, двоичное представление которого является матрично-векторным произведением Ax. Все арифметические действия при этом выполняются над полем $\boldsymbol{GF(2)}$: все значения равны либо нулю, либо единице, при этом применимы все правила сложения и умножения, за единственным исключением: $1 + 1 = 0$. Арифметику над полем $GF(2)$ можно рассматривать как обычную целочисленную арифметику, с тем отличием, что при этом рассматривается только один младший бит.

В качестве примера для множества $S_2 = \{0, 1, 2, 3\}$ матрица

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

определяет следующую перестановку π_A: $\pi_A(0) = 0$, $\pi_A(1) = 3$, $\pi_A(2) = 2$, $\pi_A(3) = 1$. Чтобы понять, почему $\pi_A(3) = 1$, заметим, что (при работе в $GF(2)$)

$$\begin{aligned}\pi_A(3) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \cdot 1 + 0 \cdot 1 \\ 1 \cdot 1 + 1 \cdot 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} ,\end{aligned}$$

что представляет собой двоичное представление единицы.

В оставшейся части данной задачи мы работаем над полем $GF(2)$, и все элементы матриц и векторов являются либо нулем, либо единицей. Определим ранг 0-1-матрицы (матрицы, в которой каждый элемент является либо нулем, либо единицей) над полем $GF(2)$ так же, как и для обычной матрицы, только при этом все арифметические действия, определяющие линейную независимость, выполняются над $GF(2)$. Определим ***диапазон*** (range) 0-1-матрицы A размером $n \times n$ как

$$\mathcal{R}(A) = \{y : y = Ax \text{ для некоторого } x \in S_n\} ,$$

так что $\mathcal{R}(A)$ представляет собой множество чисел из S_n, которые можно получить умножением каждого значения x из S_n на A.

а. Докажите, что если r — ранг матрицы A, то $|\mathcal{R}(A)| = 2^r$. Заключите отсюда, что A определяет перестановку S_n только тогда, когда A имеет полный ранг.

Для заданной матрицы A размером $n \times n$ и заданного значения $y \in \mathcal{R}(A)$ определим ***прообраз*** (preimage) y как

$$\mathcal{P}(A, y) = \{x : Ax = y\} ,$$

так что $\mathcal{P}(A, y)$ представляет собой множество значений из S_n, которые отображаются на y при умножении на A.

б. Докажите, что если r — ранг матрицы A размером $n \times n$ и $y \in \mathcal{R}(A)$, то $|\mathcal{P}(A, y)| = 2^{n-r}$.

Пусть $0 \leq m \leq n$, и предположим, что мы разбиваем множество S_n на блоки последовательных чисел, где i-й блок состоит из 2^m чисел $i2^m, i2^m + 1, i2^m + 2, \ldots, (i+1)2^m - 1$. Для любого подмножества $S \subseteq S_n$ определим $\mathcal{B}(S, m)$ как множество блоков S_n размером 2^m, содержащих некоторый элемент S. В качестве примера при $n = 3$, $m = 1$ и $S = \{1, 4, 5\}$ множество $\mathcal{B}(S, m)$ состоит из блоков 0 (поскольку 1 входит в нулевой блок) и 2 (поскольку и 4, и 5 находятся в блоке 2).

в. Пусть r — ранг нижней левой подматрицы A размером $(n - m) \times m$, т.е. матрицы, полученной путем пересечения $n - m$ нижних строк и m левых столбцов A. Пусть S — произвольный блок S_n размером 2^m и пусть $S' = \{y :$

$y = Ax$ для некоторого $x \in S\}$. Докажите, что $|\mathcal{B}(S', m)| = 2^r$ и что для каждого блока в $\mathcal{B}(S', m)$ на этот блок отображаются ровно 2^{m-r} чисел из S.

Поскольку умножение нулевого вектора на любую матрицу дает нулевой вектор, множество перестановок S_n, определяемое умножением на 0-1-матрицы размером $n \times n$ с полным рангом над полем $GF(2)$, не может включать все перестановки S_n. Расширим класс перестановок, определяемых матрично-векторным умножением, путем включения аддитивного члена, так что $x \in S_n$ отображается на $Ax + c$, где c — n-битовый вектор, а сложение выполняется над полем $GF(2)$. Например, когда

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

и

$$c = \begin{pmatrix} 0 \\ 1 \end{pmatrix} ,$$

мы получаем следующую перестановку $\pi_{A,c}$: $\pi_{A,c}(0) = 2$, $\pi_{A,c}(1) = 1$, $\pi_{A,c}(2) = 0$, $\pi_{A,c}(3) = 3$. Назовем перестановку, отображающую $x \in S_n$ на $Ax + c$ для некоторой 0-1-матрицы A размером $n \times n$ с полным рангом и некоторого n-битового вектора c, ***линейной перестановкой*** (linear permutation).

г. Воспользуйтесь комбинаторными методами, чтобы показать, что количество линейных перестановок S_n гораздо меньше числа перестановок S_n.

д. Приведите пример значения n и перестановки S_n, которую нельзя получить никакой линейной перестановкой. (*Указание:* подумайте, как для заданной перестановки умножение матрицы на вектор связано со столбцами матрицы.)

Заключительные замечания

Много информации о матрицах можно найти в учебниках по линейной алгебре. В особенности хочется отметить книги Стрэнга (Strang) [321, 322].

Литература

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(5):1259–1263, 1962.

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[11] Miklós Ajtai, Nimrod Megiddo, and Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 473–482, 1995.

[12] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[13] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[14] Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201–204, 1990.

[15] Arne Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[16] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996.

[17] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.

[18] Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, second edition, 1967.

[19] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[20] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems.* PhD thesis, University of California, Berkeley, 1994.

[21] Sanjeev Arora. The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 337–348, 1998.

[22] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[23] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing Company, 1997.

[24] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook.* CRC Press, 1999.

[25] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 1999.

[26] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. *ACM Transactions on Graphics*, 26(3), article 10, 2007.

[27] Sara Baase and Alan Van Gelder. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, third edition, 2000.

[28] Eric Bach. Private communication, 1989.

[29] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.

[30] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory—Volume I: Efficient Algorithms.* The MIT Press, 1996.

[31] David H. Bailey, King Lee, and Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[32] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007.

[33] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[34] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[35] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, and Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1(1):53–64, 1988.

[36] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[37] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[38] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[39] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[40] Samuel W. Bent and John W. John. Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.

[41] Jon L. Bentley. *Writing Efficient Programs*. Prentice Hall, 1982.

[42] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[43] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[44] Daniel Bienstock and Benjamin McClosky. Tightening simplex mixed-integer sets with guaranteed bounds. *Optimization Online*, July 2008.

[45] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, second edition, 1986.

[46] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1989. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-463.

[47] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[48] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 1995.

[49] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[50] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[51] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[52] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[53] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[54] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[55] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

[56] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer, 1993.

[57] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[58] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[59] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[60] Joseph Cheriyan and Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203–226, 1995.

[61] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[62] Boris V. Cherkassky and Andrew V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[63] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[64] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.

[65] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[66] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer, 1974.

[67] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[68] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[69] V. Chvátal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[70] Cilk Arts, Inc., Burlington, Massachusetts. *Cilk++ Programmer's Guide*, 2008. Available at `http://www.cilk.com/archive/docs/cilk1guide`.

[71] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[72] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165):297–330, 1984.

[73] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[74] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[75] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[76] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6(3):169–180, 1993.

[77] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[78] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1998.

[79] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Games*, volume 9, pages 25–52. University of Maastricht, Netherlands, 2001.

[80] Paolo D'Alberto and Alexandru Nicolau. Adaptive Strassen's matrix multiplication. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 284–292, June 2007.

[81] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, 2008.

[82] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit keys. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 142–151, January 2004.

[83] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006.

[84] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979.

[85] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[86] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[87] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[88] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[89] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[90] John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333–352, 1984.

[91] Dorit Dor, Johan Håstad, Staffan Ulfberg, and Uri Zwick. On lower bounds for selecting the median. *SIAM Journal on Discrete Mathematics*, 14(3):299–311, 2001.

[92] Dorit Dor and Uri Zwick. Selecting the median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999.

[93] Dorit Dor and Uri Zwick. Median selection requires $(2 + \epsilon)n$ comparisons. *SIAM Journal on Discrete Mathematics*, 14(3):312–325, 2001.

[94] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.

[95] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[96] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[97] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[98] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1987.

[99] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[100] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, 1971.

[101] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[102] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[103] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.

[104] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[105] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7(12):701, 1964.

[106] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, 1972.

[107] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[108] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[109] Lestor R. Ford, Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, 1959.

[110] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[111] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[112] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[113] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[114] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[115] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[116] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[117] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[118] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[119] Harold N. Gabow, Z. Galil, T. Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[120] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[121] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[122] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997.

[123] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997.

[124] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, 1992.

[125] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[126] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.

[127] Michael R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150, 1972.

[128] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[129] Saul Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, fourth edition, 1975.

[130] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[131] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[132] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

[133] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[134] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 144–191. PWS Publishing Company, 1997.

[135] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[136] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[137] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[138] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, Lászlò Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 101–164. Springer, 1990.

[139] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[140] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, editors, *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pages 73–170. Elsevier Science Publishers, 1989.

[141] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[142] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[143] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[144] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

[145] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[146] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[147] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2001.

[148] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[149] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[150] Ronald L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[151] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[152] David Gries. *The Science of Programming*. Springer, 1981.

[153] M. Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[154] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[155] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[156] H. Halberstam and R. E. Ingram, editors. *The Mathematical Papers of Sir William Rowan Hamilton*, volume III (Algebra). Cambridge University Press, 1967.

[157] Yijie Han. Improved fast integer sorting in linear space. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 793–796, 2001.

[158] Yijie Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest path. *Algorithmica*, 51(4):428–434, 2008.

[159] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[160] Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3):86–95, 2000.

[161] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, May 1965.

[162] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[163] Monika R. Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672, 1995.

[164] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[165] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222–250, 2000.

[166] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, 1990.

[167] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[168] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[169] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[170] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6(3):243–254, 1983.

[171] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[172] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27(3):713–721, 1956.

[173] Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer, 1987.

[174] Micha Hofri. *Analysis of Algorithms*. Oxford University Press, 1995.

[175] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[176] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, third edition, 2006.

[177] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[178] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[179] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[180] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.

[181] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[182] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[183] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[184] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[185] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, article 32, 1996.

[186] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[187] E. J. Isaac and R. C. Singleton. Sorting by address calculation. *Journal of the ACM*, 3(3):169–174, 1956.

[188] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[189] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[190] David S. Johnson. The NP-completeness column: An ongoing guide--The tale of the second prover. *Journal of Algorithms*, 13(3):502–524, 1992.

[191] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[192] Richard Johnsonbaugh and Marcus Schaefer. *Algorithms*. Pearson Prentice Hall, 2004.

[193] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics–Doklady*, 7(7):595–596, 1963. Translation of an article in *Doklady Akademii Nauk SSSR*, 145(2), 1962.

[194] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[195] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[196] Howard Karloff. *Linear Programming*. Birkhäuser, 1991.

[197] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[198] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[199] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1-3):165–201, 1991.

[200] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[201] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15(2):434–437, 1974.

[202] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[203] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[204] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, second edition, 1997.

[205] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287–299, 1986.

[206] Philip N. Klein and Neal E. Young. Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms*, pages 34-1–34-19. CRC Press, 1999.

[207] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[208] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Third edition, 1997.

[209] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969. Third edition, 1997.

[210] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973. Second edition, 1998.

[211] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[212] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–23, 1976.

[213] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[214] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[215] Bernhard Korte and László Lovász. Mathematical structures underlying greedy algorithms. In F. Gecseg, editor, *Fundamentals of Computation Theory*, volume 117 of *Lecture Notes in Computer Science*, pages 205–209. Springer, 1981.

[216] Bernhard Korte and László Lovász. Structural properties of greedoids. *Combinatorica*, 3(3–4):359–374, 1983.

[217] Bernhard Korte and László Lovász. Greedoids—A structural framework for the greedy algorithm. In W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984.

[218] Bernhard Korte and László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984.

[219] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer, 1992.

[220] David W. Krumme, George Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[221] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[222] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[223] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[224] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[225] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[226] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, 1985.

[227] Tom Leighton. Notes on better master theorems for divide-and-conquer recurrences. Class notes. Available at `http://citeseer.ist.psu.edu/252350.html`, October 1996.

[228] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[229] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, October 2007.

[230] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[231] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42. Springer, 1993.

[232] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[233] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. In Russian.

[234] Anany Levitin. *Introduction to the Design & Analysis of Algorithms*. Addison-Wesley, 2007.

[235] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, second edition, 1998.

[236] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[237] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[238] László Lovász and Michael D. Plummer. *Matching Theory*, volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.

[239] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[240] Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.

[241] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[242] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[243] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.

[244] Ernst W. Mayr, Hans Jürgen Prömel, and Angelika Steger, editors. *Lectures on Proof Verification and Approximation Algorithms*, volume 1367 of *Lecture Notes in Computer Science*. Springer, 1998.

[245] C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[246] M. D. McIlroy. A killer adversary for quicksort. *Software—Practice and Experience*, 29(4):341–344, 1999.

[247] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer, 1984.

[248] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer, 1984.

[249] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer, 1984.

[250] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[251] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[252] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[253] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[254] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[255] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[256] Louis Monier. *Algorithmes de Factorisation D'Entiers*. PhD thesis, L'Université Paris-Sud, 1980.

[257] Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[258] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[259] Rajeev Motwani, Joseph (Seffi) Naor, and Prabhakar Raghavan. Randomized approximation algorithms in combinatorial optimization. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 11, pages 447–481. PWS Publishing Company, 1997.

[260] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[261] J. I. Munro and V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151–160, 1996.

[262] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[263] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.

[264] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, second edition, 1998.

[265] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. *Signals and Systems*. Prentice Hall, second edition, 1997.

[266] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1):109–129, 1997.

[267] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 1998.

[268] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[269] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[270] Michael S. Paterson. Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pages 368–379, 1996.

[271] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 232–240, 2006.

[272] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 555–564, 2007.

[273] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.

[274] Steven Phillips and Jeffery Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[275] J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15(3):331–334, 1975.

[276] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10. Springer, 1993.

[277] Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981.

[278] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.

[279] William K. Pratt. *Digital Image Processing*. John Wiley & Sons, fourth edition, 2007.

[280] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[281] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, second edition, 2002.

[282] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[283] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[284] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[285] Paul W. Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.

[286] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[287] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[288] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[289] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, 1997.

[290] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[291] Edward M. Reingold, Jürg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977.

[292] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited. *Information Processing Letters*, 64(5):217–223, 1997.

[293] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*, volume 126 of *Progress in Mathematics*. Birkhäuser, second edition, 1994.

[294] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. See also U.S. Patent 4,405,829.

[295] Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26–29, 1955.

[296] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

[297] Salvador Roura. An improved master theorem for divide-and-conquer recurrences. In *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 449–459. Springer, 1997.

[298] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.

[299] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[300] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976.

[301] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[302] Alexander Schrijver. Paths and flows—A historical survey. *CWI Quarterly*, 6(3):169–183, 1993.

[303] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[304] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[305] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.

[306] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[307] Raimund Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[308] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[309] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, second edition, 2001.

[310] Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401–419, 1994.

[311] Michael I. Shamos and Dan Hoey. Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 208–215, 1976.

[312] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[313] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In William Cook, László Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.

[314] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614, 1999.

[315] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, second edition, 2006.

[316] Steven S. Skiena. *The Algorithm Design Manual*. Springer, second edition, 1998.

[317] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[318] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[319] Joel Spencer. *Ten Lectures on the Probabilistic Method*, volume 64 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1993.

[320] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

[321] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[322] Gilbert Strang. *Linear Algebra and Its Applications*. Thomson Brooks/Cole, fourth edition, 2006.

[323] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[324] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[325] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[326] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[327] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[328] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[329] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[330] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.

[331] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[332] George B. Thomas, Jr., Maurice D. Weir, Joel Hass, and Frank R. Giordano. *Thomas' Calculus*. Addison-Wesley, eleventh edition, 2005.

[333] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[334] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[335] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[336] Richard Tolimieri, Myoung An, and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer, second edition, 1997.

[337] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.

[338] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[339] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[340] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science Publishers and the MIT Press, 1990.

[341] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[342] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.

42 Зак. 3726

[343] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[344] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568–581, 1997.

[345] Hao Wang and Bill Lin. Pipelined van Emde Boas tree: Algorithms, analysis, and applications. In *26th IEEE International Conference on Computer Communications*, pages 2471–2475, 2007.

[346] Antony F. Ware. Fast approximate Fourier transforms for irregularly spaced data. *SIAM Review*, 40(4):838–856, 1998.

[347] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[348] Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995.

[349] Mark Allen Weiss. *Data Structures and Problem Solving Using C++*. Addison-Wesley, second edition, 2000.

[350] Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, third edition, 2006.

[351] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, third edition, 2007.

[352] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, second edition, 2007.

[353] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57(3):509–533, 1935.

[354] Herbert S. Wilf. *Algorithms and Complexity*. A K Peters, second edition, 2002.

[355] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7(6):347–348, 1964.

[356] Shmuel Winograd. On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, volume 3, pages 283–288, 1970.

[357] Andrew C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780–787, 1981.

[358] Chee Yap. A real elementary approach to the master recurrence and generalizations. Unpublished manuscript. Available at `http://cs.nyu.edu/yap/papers/`, July 2008.

[359] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.

[360] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. Chapman & Hall/CRC Press, 31st edition, 2003.

Предметный указатель

Б

В

Е

Ж

З

И

Й

К

Р

С

Т

У

АЛГОРИТМЫ
ПОСТРОЕНИЕ И АНАЛИЗ

ТРЕТЬЕ ИЗДАНИЕ

Томас Кормен, Чарльз Лейзерсон, Рональд Ривест и Клиффорд Штайн

Ряд книг, посвященных алгоритмам, отличается строгостью изложения материала, но страдает определенной неполнотой; другие книги охватывают огромный объем материала, но недостаточно строго излагают его. Эта книга удачно объединяет в себе полноту охвата и строгость изложения. В ней описаны самые разнообразные алгоритмы, сочетается широкий диапазон тем с глубиной и полнотой изложения; при этом изложение доступно для читателей самого разного уровня подготовки. Каждая глава книги относительно самодостаточна и может использоваться в качестве отдельной темы для изучения. Алгоритмы описаны простым языком и с применением псевдокода, который понятен любому, кто хоть в небольшой степени знаком с программированием, а пояснения принципов их работы даны без излишней математической строгости и требуют лишь элементарных знаний.

Первое издание данной книги давно стало стандартным справочным руководством для профессионалов и учебным пособием для студентов университетов. Второе издание было дополнено новыми главами, раскрывающими такие темы, как вероятностный анализ и рандомизированные алгоритмы, линейное программирование. Третье издание также существенно дополнено и пересмотрено. В него вошли две совершенно новые главы, посвященные деревьям ван Эмде Боаса и многопоточным алгоритмам, а глава, посвященная рекуррентности, существенно расширена. Изменена подача такого материала, как динамическое программирование и жадные алгоритмы, и введено новое понятие потока, основанного на ребрах, в материале о транспортных сетях. В третье издание также было добавлено множество новых упражнений и задач.

Томас Кормен — профессор информатики в колледже Дартмута и бывший директор Института литературы и риторики Дартмутского колледжа. **Чарльз Лейзерсон** — профессор информатики и электротехники в Массачусетсском технологическом институте, где также работает и профессор **Рональд Ривест**. **Клиффорд Штайн** — профессор организации производства и исследования операций в Колумбийском университете.

"*В свете взрывного роста количества данных и распространения вычислительных приложений эффективные алгоритмы востребованы в еще большей степени, чем ранее. Эта прекрасно написанная, тщательно продуманная и организованная книга является отличным введением в разработку и анализ алгоритмов. Первая ее половина представляет собой эффективный учебник теории алгоритмов, а вторая в большей степени предназначена для научных работников и любознательных студентов, которые хотели бы получить дополнительные знания об этой интересной науке*".

Шан-Хуа Тенг, Университет Южной Каролины

"*Это настоящая библия в указанной области, исчерпывающий учебник, охватывающий весь спектр современных алгоритмов: от быстрых алгоритмов и структур данных до алгоритмов с полиномиальным временем работы для решения очень сложных задач, от классических алгоритмов теории графов до специализированных алгоритмов поиска подстрок, вычислительной геометрии и теории чисел. Нельзя не упомянуть появившиеся в третьем издании деревья ван Эмде Боаса и многопоточные алгоритмы, важность которых постоянно увеличивается*".

Дэниел Шпильман, факультет информатики Йельского университета

"*Как преподаватель и исследователь в области алгоритмов с более чем двадцатилетним стажем, могу с уверенностью утверждать, что книга Кормена — лучший из встречавшихся мне учебников. Это умный, энциклопедичный и современный подход к изучению алгоритмов; наш факультет продолжит использовать эту книгу как в качестве учебника для студентов и аспирантов, так и в качестве рекомендуемого справочного пособия*".

Габриэль Робинс, факультет информатики Университета Виргинии



Издательский дом "Вильямс"
www.williamspublishing.com

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142
http://mitpress.mit.edu

ISBN 978-5-8459-1794-2

